MB0122–00

68xxx v9.0

InterTools C Compiler/Assembler Reference Manual

TASKING

Quality Development Tools Worldwide

A publication of
TASKING
Documentation Department
Copyright © 1997 TASKING, Inc.

All rights reserved. Reproduction in whole or part is prohibited without the written consent of the copyright owner.

The following trademarks are acknowledged:

DOS16M and DOS/4GW are trademarks of Rational Systems, Inc.
FLEXlm is a registered trademark of Globetrotter Software, Inc.
HP and HP–UX are trademarks of Hewlett–Packard Co.
IBM is a trademark of International Business Machines Corp.
Motorola is a trademark of Motorola, Inc.
MS–DOS and Windows are registered trademarks of Microsoft Corporation.
Solaris is a trademark of Sun Microsystems, Inc.
UNIX is a registered trademark of X/Open Company, Ltd.
VAX and VMS are trademarks of Digital Equipment Corporation.

All other trademarks are property of their respective owners.

Data subject to alteration without notice.

E–mail: support@tasking.com
WWW: http://www.tasking.com

The information in this document has been carefully reviewed and is believed to be accurate and reliable. However, TASKING assumes no liabilities for inaccuracies in this document. Furthermore, the delivery of this information does not convey to the recipient any license to use or copy the software or documentation, except as provided in an executed license agreement covering the software and documentation.

TASKING reserves the right to change specifications embodied in this document without prior notice.

Table of Contents

CHAPTER 1 1–1

Introduction 1–1

1.1 InterTools Documentation 1–1

1.1.1 The InterTools Manuals 1–3

1.1.2 Documentation Conventions 1–4

CHAPTER 2 2–1

Runtime Library 2–1

2.1 Introduction 2–1

2.2 System Initialization 2–3

2.3 I/O System 2–7

2.4 Time Functions 2–9

2.4.1 Time Conversion Routines 2–9

2.4.2 Low-level Time/Timer Routines 2–9

2.5 Storage Allocation 2–11

2.6 Support for the M68302ADS Development System 2–11

2.7 Support for the M68340BCC Development System 2–12

2.8 Support for the M68360QUADS Development System 2–13

2.9 Modifying the Libraries 2–14

2.9.1 Integrating New Routines Into an Existing Library 2–15

2.10 Library Object Modules 2–17

2.11 Summary of Library Routines 2–18

2.11.1 Standard Functions 2–18

2.11.2 Mathematical Functions 2–20

2.11.3 Standard I/O Functions 2–21

2.11.4 String Manipulation Functions 2–23

2.11.5 Non–local Goto Functions 2–25

2.11.6 Date and Time Routines 2–26

2.12 ASCII Character Set Macros and Functions 2–26

2.12.1 Global Definitions 2–28

2.12.2 Compile–time Assertions 2–28
2.12.3 Formatting of Numeric Values 2–29
2.12.4 Variable Length Argument List Access 2–29
2.12.5 Signal Handling 2–30
2.12.6 C Library Extensions 2–30
2.13 Run Time Library Routines 2–32

CHAPTER 3 3–1
The Assembly Language Reference 3–1
3.1 Preface 3–1
3.2 Related Publications 3–2
3.3 Using Assembly Language 3–4
3.4 Elements of Assembly Language 3–4
3.5 Notation 3–6

CHAPTER 4 4–1
Source Program Coding 4–1
4.1 Introduction 4–1
4.2 Comments 4–2
4.3 Source Line Format 4–3
4.3.1 Label Field 4–3
4.3.2 Operation Field 4–4
4.3.3 Operand Field 4–7
4.3.4 Comment Field 4–7
4.4 Symbols 4–7
4.4.1 Symbol Syntax 4–8
4.4.2 Symbol Definition Classes 4–8
4.4.3 User–Defined Labels 4–9
4.4.4 Location Counter Symbol "*" 4–10
4.5 Constants 4–10
4.5.1 Integer Constants 4–10
4.5.2 Character Constants 4–12
4.5.3 Floating Point Constants 4–13
4.6 Registers 4–14
4.7 Operators 4–25
4.8 Expressions 4–27
4.9 Addressing Modes 4–31

4.9.1 Register Direct Modes 4–37
4.9.2 Immediate Data 4–53
4.9.3 Notes on 68020–plus and CPU32 Addressing Modes 4–53
4.9.4 Notes on Addressing Options 4–59
4.10 Instruction Mnemonics 4–67
4.10.1 Arithmetic Operations 4–67
4.10.2 Move Instruction 4–70
4.10.3 Compare and Check Instructions 4–71
4.10.4 Logical Operations 4–73
4.10.5 Shift Operations 4–73
4.10.6 Bit Operations 4–75
4.10.7 Set on Condition 4–76
4.10.8 Branch Operations 4–76
4.10.9 Jump Operations 4–79
4.10.10 Decrement and Branch on Condition (DBcc) 4–80
4.10.11 Move Multiple Registers 4–81
4.10.12 Load Effective Address 4–83
4.10.13 Sign Extend Instructions 4–83
4.10.14 Move to/from Control Register 4–84
4.10.15 Move to/from Address Space 4–87
4.10.16 Bit Fields and Instructions (68020–plus only) 4–89
4.10.17 Bounds Check Instructions 4–95
4.10.18 Packed Decimal (68020–plus only) 4–97
4.10.19 Module Instructions (68020 only) 4–98
4.10.20 Trap on Condition Code 4–99
4.10.21 Compare and Swap with Operand 4–100
4.10.22 Flush Entry in the Address Translation Cache (ATC) (68030 only) 4–102
4.10.23 Flush Address Translation Cache (ATC) Entries (68040 and 68060 only) 4–103
4.10.24 Load an Address Translation Cache (ATC) Entry (68030 only) 4–103
4.10.25 Move to/from MMU Registers (68030 only) 4–104
4.10.26 Move to/from ACU Registers (68EC030 only) 4–105
4.10.27 Test a Logical Address (68030 only) 4–106
4.10.28 Test a Logical Address (68EC030 only) 4–107
4.10.29 Test a Logical Address (68040 only) 4–108
4.10.30 Test a Logical Address (68EC040 only) 4–108

4.10.31 Invalidate Cache Lines (68040/68EC040 and 68060 only) 4–108
4.10.32 Push and invalidate Cache Lines (68040/68EC040 68060 only) 4–109
4.10.33 Move 16 Bytes Block (68040/68EC040/68060 only) 4–110
4.11 Variants on Instruction Types 4–111
4.12 Floating Point Instructions Supported by the 68040 and 68060 .. 4–113
4.12 68881/68882 Floating Point Coprocessors Instructions 4–113
4.12.1 Floating Point Condition Codes 4–114
4.12.2 Floating Point Arithmetic Operations 4–116
4.12.3 Move to Floating Point Register 4–117
4.12.4 Move Floating Point Register to Memory 4–119
4.12.5 Floating Point Compare 4–121
4.12.6 Floating Point Conditional Branch 4–122
4.12.7 Floating Point Decrement and Branch on Condition 4–123
4.12.8 Floating Point Set on Condition 4–123
4.12.9 Floating Point Trap on Condition 4–124
4.12.10 Save Coprocessor State 4–125
4.12.11 Restore Coprocessor State 4–126
4.12.12 Floating Point Functions 4–126
4.12.13 Floating Point NO–OP 4–129
4.13 68851 Paged Memory Management Unit Coprocessor Instructions 4–129
4.13.1 Paged Memory Management Unit Condition Codes 4–130
4.13.2 Branch on PMMU Condition 4–130
4.13.3 Test, Decrement, and Branch on PMMU Condition 4–131
4.13.4 Set on PMMU Condition 4–131
4.13.5 Trap on PMMU Condition 4–131
4.13.6 Invalidate Address Translation Cache (ATC) Entries 4–132
4.13.7 Invalidate Address Translation Cache (ATC) 4–133
4.13.7 Root Pointer Table (RPT) Entries 4–133
4.13.8 Load an Address Translation Cache (ATC) Entry 4–134
4.13.9 Move PMMU Register 4–134
4.13.10 PMMU Restore Function 4–135
4.13.11 PMMU Save Function 4–136
4.13.12 Get Information about Logical Address 4–136
4.13.13 Validate a Pointer 4–137

CHAPTER 5 5-1
Assembler Directives 5-1
5.1 Assembly Control 5-2
5.1.1 COMMON – Enter Named Common Section 5-2
5.1.2 END – Program End 5-4
5.1.3 INCLUDE – Include Secondary File 5-5
5.1.4 OFFSET – Define Offsets 5-6
5.1.5 ORG – Absolute Origin 5-7
5.1.6 RESERVE – Reserve storage 5-8
5.1.7 RESUME – Resume defined section 5-9
5.1.8 RORG – Relocatable ORG 5-10
5.1.9 SECTION – Relocatable Program Section 5-12
5.2 Symbol Definition 5-13
5.2.1 EQU – Equate Symbol Value 5-14
5.2.2 FEQU – Equate Floating Point Symbol Value 5-14
5.2.3 REG – Define Register List 5-15
5.2.4 SET – Set Symbol Value 5-16
5.3 Data Definition/Storage Allocation 5-16
5.3.1 COMLINE – Unimplemented 5-17
5.3.2 DC – Define Constant 5-17
5.3.3 DCB – Define Constant Block 5-20
5.3.4 DS – Define Storage 5-21
5.4 Listing Control and Output Options 5-22
5.4.1 FAIL – Programmer Generated Error 5-22
5.4.2 FORMAT/NOFORMAT – Unimplemented 5-22
5.4.3 LIST/NOLIST – Control Listing Generation 5-23
5.4.4 LLEN – Unimplemented 5-23
5.4.5 NOOBJ – Unimplemented 5-23
5.4.6 OPT – Assembler Options 5-23
5.4.7 PAGE/NOPAGE – Control Pagination 5-27
5.4.8 SPC – Space Between Source Lines 5-28
5.4.9 STTL – Set Subtitle 5-28
5.4.10 TTL – Set Title 5-29
5.5 External Symbol Controls 5-29
5.5.1 IDNT – Relocatable Identification Record 5-29
5.5.2 XDEF – External Symbol Definition 5-30
5.5.3 XREF – External Symbol Reference 5-30

5.6 Internal Assembly Controls 5–31
5.6.1 _BRINGIN Declare external symbol 5–32
5.6.2 _DEBSYM Put out debugging information 5–32
5.6.3 _DGROUP Define data group 5–32

CHAPTER 6 6–1
Macro Operations and Conditional Assembly 6–1
6.1 Macro Operations 6–1
6.1.1 Macro Definition 6–3
6.1.2 Macro Invocation 6–3
6.1.3 Macro Parameter Definition and Use 6–4
6.1.4 Labels Within Macros 6–5
6.1.5 The MEXIT Directive 6–6
6.1.6 The NARG Symbol 6–6
6.1.7 Implementation of Macro Definition 6–7
6.1.8 Implementation of Macro Expansion 6–7
6.2 Conditional Assembly 6–9
6.2.1 Conditional Assembly Structure 6–9
6.2.2 Example of Macro and Conditional Assembly Usage 6–13

CHAPTER 7 7–1
Structured Control Statements 7–1
7.1 Keyword Symbols 7–2
7.2 Syntax 7–2
7.2.1 IF Statement 7–6
7.2.2 Floating Point Structured Assembler Syntax 7–7
7.2.3 FOR Statement 7–8
7.2.4 REPEAT Statement 7–9
7.2.5 WHILE Statement 7–10
7.3 Simple and Compound Expressions 7–10
7.3.1 Simple Expressions 7–11
7.3.2 Condition Code Expressions 7–11
7.3.3 Operand Comparison Expressions 7–12
7.3.4 Compound Expressions 7–15
7.4 Source Line Formatting 7–15
7.4.1 Class 1 Symbol Usage 7–15
7.4.2 Nesting of Structured Statements 7–17
7.5 Effects on the User's Environment 7–17

CHAPTER 8 8–1
Position–independent Code 8–1
8.1 Forcing Position Independence 8–1
8.2 Base–Displacement Addressing 8–3
8.3 Base–Displacement 8–3
APPENDIX A A–1
Instruction Set Summary A–1
1 Conventions A–2
2 68000/68010/68020–plus/CPU32 Instruction Set Summary A–4
3 Floating Point Instruction Set Summary A–14
4 68851 Instruction Set Summary A–21
APPENDIX B B–1
Character Set B–1
1 Characters Recognized B–1
2 ASCII Character Set B–2

MANUAL PURPOSE AND STRUCTURE

1. Introduction

Introduces the structure and conventions of the manuals

2. Runtime Libraries

Covers installing and changing runtime libraries.

3. The Assembly Lanuage Reference

Summarizes the structure of the assembly language and gives an assembly language overview.

4. Source Program Coding

Discusses source program coding including source line format, symbols, constants, registers, operators, expressions, addressing modes, instruction mnemonics, and other instruction types.

5. Assembler Directives

Describes and gives examples of the basic forms of the most frequently used assembler directives.

6. Macro Operations and Conditional Assembly

Describes the macro and the conditional assembly capabilities of the assembler.

7. Structured Control Statements

Describes how to use structured control statements with assembly language to improve readability of assembly language.

8. Position–independent Code

Describes Forcing Position Independence, Base–Displacement Addressing, and Base–Displacement in Conjunction with Forced Position Independence.

APPENDICES

A. Instruction Set Summary

Provides a summary of the 68000 family instruction set.

B. Character Set

Contains a list of the ASCII characters recognized by the assembler.

CHAPTER 1

Introduction

This *InterTools Reference Manual* contains run–time library and assembly language information. The structure of all InterTools documentation and conventions is described in this chapter. Please refer to the *Introduction* chapter of the *InterTools Getting Started Manual* for information concerning the InterTools system and for obtaining additional help.

1.1 InterTools Documentation

The InterTools documentation is composed of the *InterTools Getting Started Manual*, the *InterTools User's Manual*, and the *InterTools Reference Manual*. Their contents are represented in the following three charts:

InterTools Getting Started Manual
1. Introduction 2. Installation Guide 3. InterTools Tutorial

InterTools User's Manual

1. Introduction
2. 68000 Family C Compiler
3. 68000 Family Assembler
4. Linking Locator
5. Formatter
6. Other Utilities:
   - Introduction
   - Librarian
   - Global Symbol Mapper
   - Symbol List Utility
   - Object Size List Utility
7. Application Notes:
   - Introduction
   - Downloading
   - Linking C and Assembly
   - Pragma Separate (Option Separate)
   - Building Libraries That Do Not Use A5
   - Position–independent Code
   - Getting the Best Code For Your Application
   - Support for the On–board Peripherals of the 68332, 68340, and 68360
   - InterTools FLEXlm Guide (Unix only)

A. Appendix: C Language Specifications

B. Appendix: Compiler Naming Conventions

C. Appendix: Compiler Runtime Conventions

D. Appendix: Object Module Formats

E. Appendix: Compiler/Assembler Driver

F. Appendix: Supplemental Reading

InterTools Reference Manual
1. Introduction
2. Run–time Library
3. The Assembly Language Reference
4. Source Program Coding
5. Assembler Directives
6. Macro Operations and Conditional Assembly
7. Structured Control Statements
8. Position–independent Code
A. Appendix: Instruction Set Summary
B. Appendix: Character Set

1.1.1 The InterTools Manuals

The *InterTools Getting Started Manual* contains an introduction to the InterTools system and customer support, an installation guide, and a tutorial which contains sample code and exercises which lead you step–by–step through the powerful features of each software tool.

The *InterTools User's Manual* includes invocation, options, and usage summaries, along with examples for each of the tools and definitions of special terminology and functions. This manual also contains additional information in the appendices on run time and naming conventions, C language extensions, and object module formats,

The *InterTools Reference Manual* provides information on the run–time libraries and the information necessary to write programs in assembly language. It contains sections on source program coding, assembler directives, macro operations, structured control statements, and position–independent code, as well as a summary of the target instruction set and character set.

1.1.2 Documentation Conventions

In this documentation set, we use a number of typefaces to demonstrate the syntax of various commands and to show sample text you might type at a terminal or observe in a file. The following is a list of these conventions.

Typewriter Font

Used for: user input/screen output

`Typewriter` font is used in examples to show what you might type at a terminal: command names, directives, switches, literal filenames, or any other text which must be typed exactly as shown. It is also used in other examples to show what you might see on a screen or in a printed listing. For example:

```
form [prog.ab] [options]
```

`form` is shown in typewriter font because it must be typed exactly as shown when you run the formatter.

```
#include "stdio.h"
main ()
{
   printf ("hello, world.\n");
}
```

The above example shows the contents of a file that you may have output to the screen.

Italics

Used for: value substitution

Italic type indicates categories of items for which you must substitute appropriate values, such as arguments or hypothetical filenames. For example:

```
-c cfn
```

`-c` is in typewriter font because it must be typed exactly as shown. *cfn* is in italics, indicating that you must substitute an appropriate value, in this case the name of a locator command file, in place of *cfn* when using this option.

We also use *italic* type throughout this documentation set to indicate directory names in explanations.

Square Brackets

Used for: optional items

Optional items in syntax, command, or option expressions appear in square brackets ('[' and ']'). As an example, the syntax of the `-w` assembler option is shown below:

```
-w [n]
```

The *n* is optional and thus is enclosed in square brackets.

Vertical Bar

Used for: indicating that only one of the items in the group may be chosen

A vertical bar separating two or more items indicates that a choice must be made among the items. For example:

```
llink prog.[ol|rmp|ln]
```

The user should select one of the filename extensions shown.

Ellipsis

Used for: indicating that one or more items are allowed

An ellipsis (...) is used after an item to indicate that more than one of that item is allowed. The syntax of the `-I` compiler option is shown below:

```
-I dir1 [dir2...]
```

The `-I` must be typed exactly as shown. A directory name must be substituted for *dir1* and after that there may optionally be any number of additional directory names.

Shaded Boxes

Used for: command line examples

Command line examples appear in shaded boxes, as below:

```
llink prog.ol
```

Bounded Boxes

Used for: listing output

Bounded boxes display listings, as below:

```
January 20, 1990 CROSS-REFERENCE: hello.c PAGE 1

i
   Def : hello.h      2
   Ref : hello.c      5

main
   Def : hello.c      2
```

Note: Due to the page width limitations of this manual, we may represent a single invocation line as two or more lines. You should, however, type the invocation on one line.

CHAPTER 2

Runtime Library

This chapter contains the following sections:

- Introduction
- System Initialization
- I/O System
- Storage Allocation
- Support for the M68302ADS Development System
- Support for the M68340BCC Development System
- Support for the M68360QUADS Development System
- Modifying the Libraries
- Library Object Modules
- Summary of Library Routines
- Run–time Library Routines

2.1 Introduction

This section identifies the parts of the run–time library which you may need to modify to integrate with your hardware and software. It also describes in some detail the nature of the required changes and how to install them in the library. Finally, this appendix describes each library routine. The overall organization of the run–time library is discussed in the *Compiler Library Organization* section in the *Linking Locator* chapter of the *InterTools User's Manual.*

The source code for most of the library routines, including the environment–dependent routines, is supplied. However, software floating point emulation routines for the 68020 and 68040 are not supplied. The 68020 routines were obtained from U.S. Software Inc, and TASKING is not free to distribute them in source form. Source for these routines can be obtained from U.S. Software if necessary. The 68040 routines that emulate the 68881 transcendental function instructions were obtained from Motorola, Inc.

The run–time library is primarily written in C, but the lowest level functions are written in assembly language.

The run–time library supplied with the release was written assuming a particular target system: the Motorola M68000 Design Module. This system includes RAM, ROM, a timer and I/O capabilities. It also provides a firmware monitor, MACSBUG, to control program execution.This system is probably similar to your target application, so the interface routines supplied can be used as models for how you might interface to your own target system.

In addition, there are routines written specifically for the M68302ADS, M68340BCC, and M68360QUADS Development Systems. These will be discussed later.

You can code your replacement routines in C or assembly language. Remember, any routines you code in assembly language must adhere to the conventions described in the *Linking C and Assembly* application note in the *InterTools User's Manual.*

As noted in the *Linking Locator* chapter in the *Intertools User's Manual*, there are multiple versions of object modules in the library. You must decide which library(ies) will be used before deciding what changes to make to the source modules.

You must supply replacement modules suitable for inclusion in your libraries. If you code a replacement module in C, you need to compile it with the appropriate options for each library you will use. However, if you code your module in assembly language, you must be sure the assembly language routine is correct for each library.

2.2 System Initialization

Three features must be provided to establish an execution environment:

- The power–on condition on your target system must somehow transfer control to user code.
- The run–time environment must be appropriately initialized before compiled code is activated.
- An appropriate action should be taken when the top–level compiled routine exits.

These functions are provided by two assembly language run–time library routines: `pmain.68k` and `exit.68k`. You can find the source for these routines in the run–time library source directory of the InterTools release.

For some targets, the initialization source file is not called pmain.68k, as shown below:

Target	Initialization File
68302 with the M68302ADS Development System	`pmn302a.68k`
68332	`pmn332.68k`
68332 with the M68332EVS Development System	`pmn332r.68k`
68030 with the VME143 Development System	`pmn030r.68k`
68340	`pmn340.68k`
68340 with the M68340BCC Development System	`pmn340b.68k`
68360	`pmn360.68k`
68360 with the M68360QUADS Development System	`pmn360b.68k`
68060 with floating points	`pmainf.68k`

When the 68000 hardware starts execution after a power on (cold start), it loads the SSP and PC registers from absolute locations 0 and 4. The 68020 behaves similarly, except that the ISP register replaces SSP. `pmain` defines eight bytes of data which are absolutely assembled at location 0. The initial PC value is the address of the `__main` routine. This address is also designated as the "start" address of the system. The initial SSP (ISP) value is 7FFC.

`pmain` performs the setup operations required by the compiled code, and then executes a long call to the external label `_main`. By default the compiler generates the global label `_main` at the start of your C `main` routine.

Of course, you can name the `main` routine whatever you like, but the initialization module `pmain` must be adjusted accordingly. The required setup operations are enumerated below:

1. Provide a stack area and initialize the USP and SSP (user and system stack pointer) registers accordingly. For the 68020, the USP, ISP, and MSP (user, interrupt, and master stack pointers) are set.
2. Initialize A6 (frame pointer) to zero.
3. Initialize A5 to point to the global data area.

   The compiler assumes that the A5 register always contains the base address of the global data area. The global data area is a group named `data`, which consists of the `idata` and `udata` segments. In a real time system where re–entrancy is necessary, a program must dynamically allocate its stack and global data area. This ensures that multiple real time tasks running the same program will use different A5 and A7 register values.

   Note: If your system dynamically allocates the data area, remember that the base address (in A5) is the actual address of the data area if its size is less than or equal to 32K, otherwise it is the address of the data area *plus 32K*. The linking locator creates a global symbol named "ldata" whose value is the size of the data group. This symbol may be useful in coding the call to dynamically allocate memory.

   Note: It is possible to configure your system so that the A5 register is not used. You must rebuild the libraries and use the command line `-sd` switch on all compilations. See the *Building Libraries That Do Not Use A5* application note in the *InterTools User's Manual* for more details.

4. In the 68020 hardware floating point library, the 68881 floating point coprocessor is initialized by setting both the floating point status register, FPSR, and floating point control register, FPCR, to zero.

5. You will have to define what happens when the user program returns (if that is possible). This decision is reflected in two places: in the `exit.68k` routine, and just after the call to `_main` in `pmain.68k`. Our sample routines contain an infinite loop. You may want your exit routine to deliver a return code and close any I/O channels.

Please refer to the initialization file appropriate to the target you are using for more information.

2.3 I/O System

You will have to change the low level routines that "put" and "get" characters to interface with the character I/O on your target system. If you do not intend to support multiple files then this is all you need to do. If you do intend to support multiple files (our sample implementation does not) then you must modify the next higher level of I/O routines (`getc` and `putc`). You must also define what a "file control block" looks like, and what it means to "open" and "close" files.

The entire I/O system assumes some underlying structure that contains a file control block of some sort; however, only the routines mentioned below actually manipulate the contents of that structure. The only things ever passed to these routines or returned by them are pointers to the file control structure.

Here is a list of the routines you will have to provide:

- `FILE* fopen (char` *`*filename,`* `char` *`*mode`*`);`

  The first argument can be something defined by your installation: maybe a port address, maybe a pointer to a string of characters. It **must** be the same size as a pointer. The second argument must definitely be a pointer to a character that specifies the mode to open the file.

  If the mode character is `r`, it is open for reading, `w` means write, and `a` means append. `fopen` returns a pointer to a file control block.

- `void fclose (FILE *stream);`

  The argument is the kind of pointer returned by `fopen`. This routine performs cleanup tasks, for example, flushing buffers.

- `int getc (FILE *stream);`

  This function returns the next character (8–bit quantity) from the given I/O stream. The result is returned in an integer variable. It must return –1 (all bits on) when it finds an end–of–file condition.

  Our `getc` calls an external assembly language routine `_getc` which does the actual input.

- `int ungetc (char c, FILE *stream);`

  Pushes the given character back into the stream. The character is returned in an integer variable. Only one "ungetc–ed" character at a time need be supported.

- `int putc (char c, FILE *stream);`

  Writes the given character onto the given file stream. The character written is returned in an integer variable.

  Our `putc` calls an external assembly language routine `_putc` which does the actual output.

In addition you must define three global variables of type `FILE *`: `stdin, stdout, stderr`. These represent the default input, output, and error reporting I/O streams.

Sample code to perform these functions is provided in the following library modules:

```
Source          Object
aputc.68k   →   aputc.ln
aputcl.68k  →   aputcl.ln
agetc.68k   →   agetc.ln
putc.c      →   putc.ln, putc.lln
getc.c      →   getc.ln, getc.lln
fopen.c     →   fopen.ln, fopen.lln
stdio.h         (include file)
```

2.4 Time Functions

The current default libraries provide all of the time conversion and low-level time/timer routines described in Section 4.12 of the ANSI Standard. However, all low-level timer functions return ANSI values, stating that the timer function is not implemented.

If your application requires current low-level time/timer information, in addition to time conversion, you must modify the low-level time/timer routines to use the time hardware. These modifications are discussed below.

2.4.1 Time Conversion Routines

The `gmtime` routine is the only time conversion routine that returns the value of `(struct tm *) NULL`, which requires a low-level time function. You should modify this routine so that it returns the current UTC time, as prescribed in ANSI.

2.4.2 Low-level Time/Timer Routines

The `clock` in the `time.c` routine currently returns the value of `(clock_t)-1`. You should modify this routine so that it returns the elapsed clock count, as prescribed in ANSI.

The `time` in the `time.c` routine currently returns the value of `(time_t)-1`. You should modify this routine so that it returns the current calendar time, as prescribed in ANSI.

2.5 Storage Allocation

The library storage allocation routines request "system" storage when they do not possess enough free storage to satisfy an allocation request. The routine which provides system storage is called `_alloc`.

```
char * _alloc (size_t request, size_t *given);
```

The first parameter is an integer: the number of words requested. The second parameter is a pointer to an integer. The routine returns the null pointer if it cannot provide at least as many words as were requested. Otherwise, it returns a pointer to a chunk of storage and sets the integer pointed to by the second parameter to the number of words actually allocated. This might be more than was actually requested.

A sample implementation is provided by the `xalloc` module for most targets. It implements a 4K heap.

```
xalloc.c → xalloc.ln, xalloc.lln
```

2.6 Support for the M68302ADS Development System

There are two libraries that support the 68302 target with the M68302ADS Development System. `lib302ap` contains modules to support the M68302ADS with parallel I/O; `lib302at` contains modules to support the M68302ADS with trap–based I/O. The source files written specifically for this environment have the characters `302a` in their names.

In addition to the extra source files, a locator command file, `ads302.cmd`, is supplied to specify the memory map of the M68302ADS card. This file results in the following:

- A `MEMORY` command defines the maximum size of memory on the M68302ADS as 512 kilobytes of RAM.
- A `RESERVE` command ensures that the first 0x4000 bytes are reserved for use by the monitor, `bug302`.
- The startup module, defined in `pmn302a.68k`, is in segment `init` and is located at address 0x4000.
- The remaining segments are placed in memory starting at address 0x4080.
- The segment `S_end_project` is used by the dynamic memory allocator to indicate the end of used memory.

See `ads302.cmd` for more details.

2.7 Support for the M68340BCC Development System

The `lib340b` library supports the 68340 target with the M68340BCC Development System. The source files written specifically for this environment have the characters `340b` in their names.

In addition to the extra source files, a locator command file, `bcc340.cmd`, is supplied to specify the memory map of the M68340BCC card. This file results in the following:

- A `MEMORY` command defines the maximum size of memory on the M68340BCC as 64 kilobytes of RAM.
- A `RESERVE` command ensures that the first 0x3000 bytes are reserved for use by the monitor program, 340bug.
- The startup module, defined in `pmn340b.68k`, is in segment `init` and is located at address 0x3000.
- The remaining segments are placed in memory starting at address 0x3080.
- The segment `S_end_project` is used by the dynamic memory allocator to indicate the end of used memory.

See `bcc340.cmd` for more details.

2.8 Support for the M68360QUADS Development System

The `lib360b` library supports the 68360 target with the M68360QUADS Development System. The source files written specifically for this environment have the characters `360b` in their names. The `lib360b` library also uses some M68340BCC sources files (which are M68360QUADS–compatible). These files have the characters `340b` in their names.

In addition to the extra source files, a locator command file, `quads360.cmd`, is supplied to specify the memory map of the M68360QUADS card. This file results in the following:

- A `MEMORY` command defines the maximum size of memory on the M68360QUADS as 0x4E0000 of RAM.
- A `RESERVE` command ensures that locations between 0x0 – 0x20000 and 0x21800 – 0x400000 are reserved.
- The startup module, defined in `pmn360b.68k`, is in segment `init` and is located at address 0x400000.
- The segment `S_end_project` is used by the dynamic memory allocator to indicate the end of used memory.

See `quads360.cmd` for more details.

2.9 Modifying the Libraries

Once you know what your low–level routines are going to look like, you can begin editing your replacement files. When you have completed your replacement routines, refer to the following procedure to integrate your new routines into the library.

The calling convention with which the runtime library you are modifying was built affects the way you assemble or compile any new library routines. All routines in a library must use the same calling convention.

Given the number of different runtime libraries, it is possible that you may have to go through this entire procedure several times. That is, once for the each target and once for both the hardware and software floating libraries, if you are using both. The integration procedure is identical in all cases. Of course, if you never intend to use a library you need not update it.

The only time the hardware/software library pairing affects the coding of your low–level routines is with the system initialization routine. You may need two slightly different `pmain` routines: one which contains instructions to initialize the 68881/68882 floating point coprocessor and one which does not. Of course, the one which initializes the coprocessor goes in the hardware floating point library.

From now on, we will describe the process of integrating with the library as if only the 68000 library exists.

2.9.1 Integrating New Routines Into an Existing Library Without Using `make` on Unix Hosts

1. Save the original versions of the sources and object modules for all the routines you will change.
2. Copy your new source files into the runtime library directory.
3. Assemble any new assembly language source modules.
4. Link each of the resulting `.o1` object module files with itself to produce the new `.ln` files. Use the `llink` utility and supply the `-lo`, `-o` and `-w` switches.
5. Compile all the new C source modules supplying the `-L` switch.
6. Link each of the resulting object modules with itself. Supply the `-lo`, `-o` and `-w` switches to the `llink` utility to produce a `.ln` module for each new object module.
7. Rename each linked C module from `- - -.ln` to `- - -.lln`.
8. Compile all the new C source modules again, but this time **without** `-L`.
9. Link each of the resulting object modules with itself. Supply the `-lo`, `-o` and `-w` switches to the `llink` utility to produce a `.ln` module for each new object module.

10. Update the library index files using the librarian utility. This process is described in more detail below.

To update each library, you must use the librarian to:

- Delete the old object module, unless the new one has the same name.
- Add/replace the appropriate new module(s).

Example: Updating _alloc

Suppose you have coded a replacement for `xalloc.c` and your replacement routine is also called `xalloc.c`. You have compiled and linked your routine twice, once with the `-L` switch and once without, producing `xalloc.lln` and `xalloc.ln`.

Enter the following librarian invocations:

```
libr xalloc.ln -v -L lib000
libr xalloc.ln -v -L lib000.nf
libr xalloc.lln -v -L lib000.l
libr xalloc.lln -v -L lib000.lnf
```

Note: There is no need to delete the old library member, since the new member has the same name.

You can use the librarian utility to list all library entries.

2.10 Library Object Modules

Depending upon which runtime libraries have been installed on your system, your runtime library may include several directories containing library index files and library object modules. A single linked object module may be named in more than one library index file in its directory. For ease in building and modifying runtime libraries, a standard naming convention is used for the linked object modules.

When you modify a library source, you must rebuild all the corresponding object modules. This may be done by following the steps outlined above, in the section *Modifying the Libraries*.

Not all sources are compiled into all possible suffix combinations; you need only replace the ones which exist. For example, the `.in` suffix only applies to two modules: `xprintf.c` and `xscanf.c`. This special suffix is produced from compiling these modules with the `-P NO_FP_IO` option. This option defines a preprocessor variable which causes the compiler to exclude the code printing or scanning floating point data. The resulting module can only be used in a "no–floats" library index file.

The following table summarizes the correct compiler options to supply when recompiling a library module and the meaning of the object module's utilities:

Suffix	Meaning	Compiler Switches
`.ln`	Default	
`.in`	No Floats	`-P NO_FP_IO`
`.lln`	Long Ints	`-L`
`.iln`	Long Ints, No Floats	`-L -P NO_FP_IO`

2.11 Summary of Library Routines

Runtime library routines can be accessed from C source code, via `#include` statements to resolve references such as simple math and I/O functions. The library routines follow the ANSI specification. In addition, InterTools has its own set of library routines described in the include file `extended.h`.

2.11.1 Standard Functions

The standard functions are described in the ANSI C specification. Their external declarations are available in the library include file `stdlib.h`. A table summarizing the standard routines appears below:

Standard Functions	
Name	**Definition**
`abort`	terminate program
`abs`	absolute value
`atexit`	register functions to be called at normal program termination
`atof`	string to double conversion
`atoi`	string to integer conversion
`atol`	string to long integer conversion
`bsearch`	search an array of objects
`calloc`	allocate and zero dynamic storage
`div`	compute integer quotient and remainder
`exit`	terminate a process

Standard Functions	
Name	**Definition**
`free`	free previously allocated storage
`getenv`	get an environment variable
`labs`	long integer absolute value
`ldiv`	compute long quotient and remainder
`malloc`	allocate but do not zero dynamic storage
`mblen`	return multi–byte character length
`mbstowcs`	convert multi–byte string to wide–char string
`mbtowc`	convert multi–byte char to wide–char char
`qsort`	sort an array of elements
`rand`	return a random number between 0 and 32767
`realloc`	change the size of an object
`srand`	reset the seed for `rand`, the random number generator
`strtod`	convert a string into a double
`strtol`	convert a string into a long integer
`strtoul`	convert a string into an unsigned long integer
`system`	pass a string to the host environment's command processor
`wcstombs`	convert wide–char string to multi–byte string
`wctomb`	convert wide–char char to multi–byte char

2.11.2 Mathematical Functions

Mathematical functions compute and return a value based on the given argument(s). Their external declarations are available in the library include file `math.h`. A table summarizing the mathematical routines appears below:

Mathematical Functions

Name	Definition
`acos`	arccosine
`asin`	arcsine
`atan`	arctangent in range $-\pi/2$ to $\pi/2$
`atan2`	arctangent of x/y in range $-\pi$ to π
`ceil`	round to more positive integer
`cos`	cosine
`cosh`	hyperbolic cosine
`exp`	exponential
`fabs`	floating point absolute value
`floor`	round to more negative integer
`fmod`	floating point modulus
`frexp`	extract fraction from exponent
`ldexp`	scale double exponent
`log`	natural logarithm
`log10`	common (base 10) logarithm
`modf`	extract fraction and integer from double
`pow`	raise x to the y power

Mathematical Functions	
Name	**Definition**
`sin`	sine
`sinh`	hyperbolic sine
`sqrt`	real square root
`tan`	tangent
`tanh`	hyperbolic tangent

2.11.3 Standard I/O Functions

The I/O system assumes some underlying structure that contains a file control block of some sort. Some files actually manipulate the contents of that structure. The include file `stdio.h` contains external declarations for these functions. A summary of standard I/O functions appears below:

Standard I/O Functions	
Name	**Definition**
`clearerr`	clear the end–of–file and error indicators
`fclose`	close the specified file
`feof`	test the end–of–file indicator for a file
`ferror`	test the error indicator for a file
`fflush`	flush output buffer
`fgetc`	read a character from the specified file

Standard I/O Functions	
Name	**Definition**
`fgetpos`	store the current value of the file position indicator
`fgets`	read a string from the specified file
`fopen`	open a file
`fprintf`	write formatted output to the specified file
`fputc`	write a character to the specified file
`fputs`	write a string to the specified file
`fread`	block read from file
`freopen`	close, then open the specified file
`fscanf`	read formatted input from specified file
`fseek`	set the file position indicator for a file
`fsetpos`	set the file position indicator for a file to a specific value
`ftell`	return the current file position indicator for a specific stream
`fwrite`	block write to file
`getc`	same as `fgetc`
`getchar`	read a character from standard input
`gets`	read a line from standard input
`perror`	map the error number in an integer expression to an error message
`printf`	write formatted output to standard output
`putc`	same as `fputc`
`putchar`	write a character to standard output

Standard I/O Functions	
Name	**Definition**
`puts`	write a string to standard output
`remove`	delete a file
`rename`	rename a file
`rewind`	rewind a file
`scanf`	read formatted input from standard input
`setbuf`	set I/O buffer
`setvbuf`	set the buffering mode
`sprintf`	write formatted output to the specified string
`sscanf`	read formatted input from the specified string
`tmpfile`	create a temporary file
`tmpnam`	return a valid, unused filename
`ungetc`	push a character back into the specified file
`vfprintf`	write formatted output to specified file using variable arguments
`vprintf`	write formatted output to standard output using variable arguments
`vsprintf`	write formatted output to the specified string using variable arguments

2.11.4 String Manipulation Functions

String manipulation functions copy and test character strings in memory. The include file string.h contains external declarations for

these functions. A summary of string manipulation functions appears in the table below:

String Manipulation Functions	
Name	**Definition**
`memchr`	search for a character in a buffer
`memcmp`	compare two buffers for lexical order
`memcpy`	copy one buffer to another
`memmove`	copy characters
`memset`	propagate a fill character throughout a buffer
`strcat`	concatenate two strings
`strchr`	scan a string for the first occurrence of a character
`strcmp`	compare two strings for lexical order
`strcoll`	compare two strings according to the current locale
`strcpy`	copy one string to another
`strcspn`	find the end of a span of characters in a set
`strerror`	map the error number to an error message
`strlen`	find the length of a string
`strncat`	concatenate two strings; append up to *n* characters
`strncmp`	compare two strings, up to *n* characters
`strncpy`	copy *n* length string
`strpbrk`	find occurrence in string of character in set
`strrchr`	scan string for the last occurrence of a character

String Manipulation Functions	
Name	**Definition**
`strspn`	find the end of a span of characters not in a set
`strstr`	find the first instance of a string
`strtok`	break string into tokens
`strxfrm`	transform string according to the current locale

2.11.5 Non–local Goto Functions

Non–local goto functions are used to define and restore an "environment." In this implementation, an "environment" consists of a set of values in the non–volatile machine registers. These functions are useful for dealing with errors and interrupts encountered in a low–level subroutine of a program. The include file `setjmp.h` contains external declarations for non–local goto functions. A summary of the functions appears below:

Non–local Goto Functions	
Name	**Definition**
`longjmp`	returns an environment established earlier by `setjmp`
`setjmp`	establishes an environment for later use by `longjmp`

2.11.6 Date and Time Routines

A summary of date and time functions appears below. The include file `time.h` contains external declarations for date and time routines. The following table summarizes date and time functions:

Date and Time Routines	
Name	**Definition**
`asctime`	convert time to a string
`clock`	return the processor time used
`ctime`	convert calendar time to local time
`difftime`	compute the difference between two times
`gmtime`	convert calendar time to Coordinated Universal Time
`localtime`	convert calendar time to local time
`mktime`	convert broken–down time into calendar time
`strftime`	put characters into an array
`time`	return the current calendar time

2.12 ASCII Character Set Macros and Functions

ASCII character set macros are of two general types: "is" and "to."

Macros with the prefix `is` take a character type parameter and evaluate to 0 or 1, acting as predicates. These macros can be used in `if`, `while` and `for` constructs. Macros with the prefix `to` convert between upper and lower–case.

The library include file `ctype.h` contains the ASCII character set macro and function definitions. A summary appears below:

ASCII Character Set Macros and Functions	
Name	**Definition**
`isalnum`	test for alphanumeric character
`isalpha`	test for alphabetic character
`iscntrl`	test for control character
`isdigit`	test for digit
`isgraph`	test for graphic character
`islower`	test for lowercase character
`isprint`	test for printing character
`ispunct`	test for punctuation character
`isspace`	test for whitespace character
`isupper`	test for uppercase character
`isxdigit`	test for hexadecimal digit
`tolower`	convert character to lowercase, if necessary
`toupper`	convert character to uppercase, if necessary

2.12.1 Global Definitions

The include file `stddef.h` contains global definitions for use in C programs. `stddef.h` defines one macro whose summary appears below:

Global Definitions	
Name	**Definition**
`offsetof`	return offset of member in structure

2.12.2 Compile–time Assertions

The include file `assert.h` defines the macro `assert` which allows compile–time testing of expressions. If the expression is false, the assert macro writes information about the call that failed on the standard error file and then aborts the program. A summary of the assert macro appears below:

Compile–time Assertions	
Name	**Definition**
`assert`	check runtime expressions

2.12.3 Formatting of Numeric Values

Two functions, `setlocale` and `localeconv`, allow the setting of defaults for the formatting of numeric values that depend on country and/or language. Their external declarations are available in the library include file `locale.h`. A table summarizing these functions appears below:

Formatting of Numeric Values	
Name	**Definition**
`localeconv`	set numeric formatting values for locale
`setlocale`	select or query the current locale

2.12.4 Variable Length Argument List Access

The argument list macros facilitate the access of items on a variable length argument list. The macros are defined in the library include file `stdarg.h` and are summarized in the table below:

Variable Length Argument List Access	
Name	**Definition**
`va_arg`	retrieve the next item on a variable length argument list.
`va_end`	finish access of variable length argument list
`va_start`	prepare to access a variable length argument list

2.12.5 Signal Handling

The library include file `signal.h` contains external declarations for dealing with internal and external events, or signals. A table summarizing these functions appears below:

Signal Handling	
Name	**Definition**
`raise`	generate a signal
`signal`	designate a function as a signal handler

2.12.6 C Library Extensions

Several extensions to the ANSI C library are available for use. These functions and macros are distributed with InterTools compilers for use in cases where their functionality may be helpful for embedded systems programming. Note however, that these functions are **not** part of the ANSI C standard, and their use may cause portability problems. The pre–processor macro _EXTENSIONS must be defined to make these external function declarations and macro definitions available. These functions are found in `extended.h`. The functions and macros are summarized in the table on the next page.

C Library Extensions	
Name	**Definition**
`__tolower`	a macro to convert character to lowercase. The character **must** be a valid uppercase ASCII letter.
`__toupper`	a macro to convert character to uppercase. The character **must** be a valid lowercase ASCII letter.
`atanh`	inverse hyperbolic tangent
`log2`	base 2 logarithm
`getl`	direct long integer input
`getw`	direct integer input
`putl`	direct long integer output
`putw`	direct integer output
`memccpy`	copy memory up to marker character

Three additional include files are distributed in ANSI C compliant InterTools compiler releases. These library include files contain definitions only, they do not contain any external function declarations. The files are: `errno.h` which contains variables used to process errors in the C language, `limits.h` which contains various defined CPU specific limitations, and `float.h` which defines various floating point hardware limits and values.

In addition, there are several include files distributed that are used only in conjunction with building libraries. Under normal circumstances, these library include files will not be used. However, if the embedded application requires changes to the run time library source, you **may** need to modify these files as required.

2.13 Run Time Library Routines

The following pages describe each of the run time routines in reference format. The descriptions are in alphabetical order by function name.

__tolower

Syntax:

```
#define _EXTENSIONS
#include <stdio.h>
#define __tolower(c) ((c) + 'a' - 'A')
```

Function: `__tolower` is a macro which converts an uppercase letter, *c*, to its lowercase equivalent. `__tolower` returns the corresponding lowercase character.

Notes: `__tolower` should only be used when *c* is known to be an uppercase letter (presumably checked via the `isupper` macro). Unlike the ANSI `tolower` function, `__tolower` is not guaranteed to return its argument if the argument is not an uppercase character.

`__tolower` is an InterTools extension.

__toupper

Syntax:

```
#define _EXTENSIONS
#include <stdio.h>
#define __toupper(c) ((c) + 'A' - 'a')
```

Function: `__toupper` is a macro which converts a lowercase letter, *c*, to its uppercase equivalent. `__toupper` returns the corresponding uppercase character.

Notes: `__toupper` should only be used when *c* is known to be a lowercase letter (presumably checked via the `islower` macro). Unlike the ANSI `toupper` function, `__toupper` is not guaranteed to return its argument if its argument is not a lowercase letter.

`__toupper` is an InterTools extension.

abort

Syntax:

```
#include <stdlib.h>
void abort(void);
```

Function: `abort` raises a `SIGABRT` condition.

abs

Syntax:

```
#include <stdlib.h>
int abs(int x);
```

Function: `abs` calculates | *x* |, the absolute value of the integer argument, *x*.

Notes: `abs` returns its input if x is the most negative int value.

acos

Syntax:

```
#include <math.h>
double acos(double x);
```

Function: `acos` computes the value whose cosine is x. The inverse cosine is in radians and in the range from zero to π.

Notes: If x is outside the range [–1,1], `acos` sets `errno` to EDOM. The return value is undefined in this case.

asctime

Syntax:

```
#include <time.h>
char * asctime(const struct tm *timeblock);
```

Function: The function `asctime` converts the time stored in **timeblock* into a 26 character string of the form:

```
Mon Jan 01 01:01:01 1990\n\0
```

`asctime` returns a pointer to the character string.

Notes: The string returned by `asctime` may be overwritten by subsequent calls to `asctime` or `ctime`.

asin

Syntax:

```
#include <math.h>
double asin(double x);
```

Function: `asin` computes the value whose sine is x. The inverse sine is in radians and is in the range from $-\pi/2$ to $\pi/2$.

Notes: If x is outside the range [–1,1], `asin` sets `errno` to EDOM. The return value is undefined in this case.

assert

Syntax:

```
#include <assert.h>
void assert(int expression);
```

Function: The `assert` macro puts diagnostics into programs. When it is executed, if *expression* is false the `assert` macro writes information about the particular call that failed, including the text of the argument, the name of the source file, and the source line number. It then aborts the program by using `abort()`.

Notes: If the preprocessor variable `NDEBUG` is defined before including `assert.h`, the `assert` macro will have no effect.

atan

Syntax:

```
#include <math.h>
double atan(double x);
```

Function: `atan` computes the value whose tangent is *x*. The inverse tangent is in radians, and is in the range from $-\pi/2$ to $\pi/2$.

atan2

Syntax:

```
#include <math.h>
double atan2(double x, double y);
```

Function: `atan2` computes the principal value of the arctangent of x/y, using the signs of both arguments to determine the quadrant of the return value. If both x and y are zero, `errno` is set to EDOM and $\pi/2$ is returned.

Notes: The return value is in radians and is in the range $-\pi$ to π. Here is a chart showing how the sign of the arguments determines the range of the return value:

Sign of arguments	Return range
$x<0$, $y<0$	$-\pi$ to $-\pi/2$
$x<0$, $y>0$	$-\pi/2$ to 0
$x>0$, $y>0$	0 to $\pi/2$
$x>0$, $y<0$	$\pi/2$ to π

atanh

Syntax:

```
#define _EXTENSIONS
#include <math.h>
double atanh(double x);
```

Function: `atanh` computes the inverse hyperbolic tangent of *x*.

Notes: `atanh` is an InterTools extension.

atexit

Syntax:

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Function: The `atexit` function causes the function pointed to by *func* to be called without arguments at normal program termination. Up to 32 functions can be remembered. `atexit` returns 0 if successful, and non–zero if it fails.

Notes: The functions are called in a last in, first out basis. Functions may be recorded more than once. Normal termination means `exit` was called.

atof

Syntax:

```
#include <stdlib.h>
double atof(const char *s);
```

Function: `atof` converts the ASCII string *s* to a double and returns the new value. The ASCII string is examined with respect to the following pattern: any number of leading white–space characters (as specified by the `isspace` function), an optional plus or minus sign, any number of decimal digits, an optional decimal point followed by any number of decimal digits, an optional "e" or "E", an optional plus or minus sign or blank, and any number of decimal digits. Input which matches this pattern is converted into the double return value. Input after the end of the pattern is ignored.

In case of error, `atof` is not required to set errno, and its return value is undefined. In this implementation, atof is implemented via `strtod`, as follows:

```
strtod(s, (char **) NULL)
```

atoi

Syntax:

```
#include <stdlib.h>
int atoi(const char *s);
```

Function: `atoi` converts the ASCII string *s* into an integer. The ASCII string is examined with respect to the following pattern: any number of leading white–space characters (as specified by the `isspace` function), an optional plus or minus sign, and any number of decimal digits. Characters which match this pattern are converted to the integer return value. Characters after the end of the pattern are ignored. `atoi` returns the converted integer value. Except in the case of errors, `atoi` is equivalent to:

```
(int) strtol(s, (char **) NULL, 10)
```

atol

Syntax:

```
#include <stdlib.h>
long atol(const char *s);
```

Function: `atol` converts the ASCII string *s* to a long integer. The ASCII string is examined with respect to the following pattern: any number of white–space characters (as defined by the `isspace` function), an optional plus or minus sign, and any number of decimal digits. Characters which match this pattern are converted to the long return value. Characters after the end of the pattern are ignored. `atol` returns the converted long integer.

Notes: Overflows are ignored. Except for the behavior on errors, `atol` is equivalent to:

```
strtol(s, (char **)NULL, 10)
```

bsearch

Syntax:

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
    size_t nummembr, size_t size,
    int (*comp)(const void *, const void *));
```

Function: `bsearch` searches an array of *nummembr* objects, the first element of which is pointed to by *base*, for an element that matches the object pointed to by *key*. The size of each array element is specified by *size*.

The function pointed to by *comp* is called with two arguments that point to the key object and to an array element, in that order. The *comp* function shall return an integer less than, equal to, or greater than zero if the *key* object is considered to be less than, equal to, or to be greater than the array element. The array should consist of all elements that compare less than, all elements that compare equal to, and all elements that compare greater than the *key* object, in that order. `bsearch` returns a pointer to the item if found, or `NULL` if not found.

calloc

Syntax:

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

Function: `calloc` allocates memory for the *nmemb* elements of size *size* and returns a pointer to the start of the allocated memory space. Allocated memory is initialized to zero.

Notes: If memory is exhausted, `calloc` returns a null pointer.

ceil

Syntax:

```
#include <math.h>
double ceil(double x);
```

Function: `ceil` computes and returns the smallest integer value greater than or equal to *x*, expressed as a double value.

clearerr

Syntax:

```
#include <stdio.h>
void clearerr(FILE *str);
```

Function: This function clears the end–of–file and error indicators for the the stream pointed to by *str*.

Notes: Users should implement this routine themselves.

clock

Syntax:

```
#include <time.h>
clock_t clock(void);
```

Function: `clock` returns the processor time used. If the processor time is unavailable, the return value is (`clock_t`)–1.

Notes: Users should implement this routine themselves. The current return value is (`clock_1`)–1.

cos

Syntax:

```
#include <math.h>
double cos(double x);
```

Function: `cos` computes the cosine of x, expressed in radians.

cosh

Syntax:

```
#include <math.h>
double cosh(double x);
```

Function: `cosh` computes the hyperbolic cosine of x.

ctime

Syntax:

```
#include <time.h>
char *ctime(const time_t *time);
```

Function: The `ctime` function converts the calendar time pointed to by *time* to local time in the form of a string.

Notes: This function is equivalent to:

`asctime(localtime(`*time*`))`

The value returned by `ctime` will be overwritten by the next call to `ctime` or `asctime`.

difftime

Syntax:

```
#include <time.h>
double difftime(time_t time1, time_t time2);
```

Function: `difftime` computes the difference between two times: *time1* and *time2*. `difftime` returns the difference in seconds, expressed as a double.

div

Syntax:

```
#include <stdlib.h>
div_t div(int numerator, int denominator);
```

Function: `div` computes the quotient and remainder of the numerator divided by the denominator. The type `div_t` is a structure that contains two components, `quot` and `rem`, both of type `int`.

When *denominator* equals zero `div` returns zeroes in *quot* and *rem*.

Notes: If the result cannot be represented, the behavior is undefined. Otherwise *quot* * *denominator* + *rem* shall equal *numerator*.

exit

Syntax:

```
#include <stdlib.h>
void exit(int status);
```

Function: `exit` caused normal program termination. `exit` never returns to its caller. First, any functions recorded by `atexit` are called, in reverse order of their presentation via `atexit`. Next, all open output streams are flushed, all open streams are closed, and all files created by the `tmpfile` function are removed. Finally, the return status is returned to the host environment.

Notes: `exit` calls `__exit` to return to the host environment. The `__exit` routine must be coded by the user.

exp

Syntax:

```
#include <math.h>
double exp(double x);
```

Function: exp computes the exponential function of *x*.

Notes: On overflow exp returns an IEEE infinity.

fabs

Syntax:

```
#include <math.h>
double fabs(double x);
```

Function: `fabs` computes |*x*|, the absolute value of the double *x*.

fclose

Syntax:

```
#include <stdio.h>
int fclose(FILE *stream);
```

Function: `fclose` flushes any buffers for the named *stream* and causes the file to be closed. The return value indicates the presence of an I/O error.

Notes: `fclose` must be implemented by the user.

feof

Syntax:

```
#include <stdio.h>
int feof(FILE *str);
```

Function: This function tests the end–of–file indicator for the stream pointed to by *str*. `feof` returns nonzero only if the end–of–file indicator is set for *str*.

Notes: `feof` must be implemented by the user.

ferror

Syntax:

```
#include <stdio.h>
int ferror(FILE *stream);
```

Function: This routine tests the error indicator for the the stream pointed to by *stream*. `ferror` returns nonzero only if the error indicator is set for *stream*.

Notes: `ferror` must be implemented by the user.

fflush

Syntax:

```
#include <stdio.h>
int fflush(FILE *stream);
```

Function: `fflush` forces out any buffered output on an I/O channel. `fflush` returns zero, if successful. If an I/O error was encountered, `fflush` sets the global integer `errno` to indicate the error code and returns negative one.

Notes: The release version of `fflush` is a stub routine which always returns zero. The standard I/O library does not buffer output.

fgetc

Syntax:

```
#include <stdio.h>
int fgetc(FILE *stream);
```

Function: `fgetc` returns the next character from the named input stream pointed to by *stream*.

fgetpos

Syntax:

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

Function: `fgetpos` stores the current value of the file position indicator for the stream pointed to by *stream* in the object pointed to by *pos*. The information stored can be used by `fsetpos` for resetting the stream to the time of the call to `fgetpos`. `fgetpos` returns zero on success; on failure `fgetpos` returns a nonzero number and sets `errno`.

Notes: `fgetpos` must be implemented by the user.

fgets

Syntax:

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

Function: `fgets` reads characters from the specified input *stream* into the string *s* until either *n*–1 characters are read, end–of–file is reached, or a newline character is read. If a newline character is read, it is retained. The stream is then terminated by a null. The associated file pointer is incremented by the number of bytes read.

Notes: If end–of–file is reached before any characters have been read, *s* remains unchanged, and `fgets` returns NULL.

floor

Syntax:

```
#include <math.h>
double floor(double x);
```

Function: `floor` computes the largest integer value less than or equal to *x*, expressed as a double value.

fmod

Syntax:

```
#include <math.h>
double fmod(double x, double y);
```

Function: `fmod` calculates the floating point remainder of *x*/*y*. That is, `fmod` returns the value *x*–*i***y*, for some integer *i* such that the result has the same sign as *x* and magnitude less than the magnitude of *y*. If *y* is zero, `fmod` returns zero under software floating point and IEEE NaN (Not a Number) under hardware floating point.

fopen

Syntax:

```
#include <stdio.h>
FILE *fopen(const char *filename,
   const char *mode);
```

Function: `fopen` opens the file *filename* and returns an associated input or output stream, depending on *mode*.

Notes: `fopen` must be implemented by the user.

fprintf

Syntax:

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format,
...);
```

Function: See `printf`.

Notes: `fprintf` is identical to `printf`, but directs its output to the specified output *stream*.

fputc

Syntax:

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

Function: `fputc` writes a character specified by *c* (converted to an unsigned char), onto the named output *stream*. `fputc` returns the character written.

fputs

Syntax:

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

Function: `fputs` copies the string *s* to the specified output *stream*. The terminating null byte is not copied. The `fputs` function returns `EOF` if a write error occurs; otherwise it returns a non–negative value.

fread

Syntax:

```
#include <stdio.h>
size_t fread(void *ptr, size_t size,
   size_t n, FILE *stream);
```

Function: `fread` reads *n* items of size *size* from the specified input *stream* into a buffer at *ptr*. `fread` returns the number of items successfully read, which may be less than *n* if a read error or end–of–file is encountered. If *size* or *n* is zero, `fread` returns zero and the contents of the array and the state of the stream remain unchanged.

free

Syntax:

```
#include <stdlib.h>
void free(void *ptr);
```

Function: `free` deallocates storage allocated by previous `malloc`, `calloc`, or `realloc` routines.

Notes: In ANSI C, `free` replaces the Unix `cfree` routine.

freopen

Syntax:

```
#include <stdio.h>
FILE *freopen(const char *filename,
   const char *mode, FILE *stream);
```

Function: `freopen` closes the specified stream and then opens it in the same way that `fopen` does. Failure to close the file is ignored. The error and end–of–file indicators for the stream are cleared. `freopen` returns a null pointer if not successful. Upon success `freopen` returns *stream*.

Notes: `freopen` must be implemented by the user. The primary use of `freopen` is to change the file associated with the standard text streams `stdin`, `stdout`, and `stderr`.

frexp

Syntax:

```
#include <math.h>
double frexp(double x, int *eptr);
```

Function: `frexp` breaks a floating point number into a normalized fraction and an integral power of two. It stores the integer in the int object pointed to by *eptr* and returns the fraction.

After calling the function

```
y = frexp(x, &n)
```

the following identity is true:

$x = y * 2^n$.

fscanf

Syntax

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format,
...);
```

Function: See `scanf`.

Notes: `fscanf` is identical to `scanf`, except that it reads its input from the specified input *stream*.

fseek

Syntax

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int
mode);
```

Function: `fseek` sets the file position indicator for the stream pointed to by *stream*. `fseek` returns zero on success and nonzero on failure.

For binary streams, `fseek` calculates the new position by adding the *offset* (in bytes) to the position specified by *mode*. The specified position is the beginning of the file if *mode* is `SEEK_SET`, the current file position if *mode* is `SEEK_CUR`, or end of file if *mode* is `SEEK_END`.

For text streams, *offset* is either zero or a value returned by a previous call to `ftell`, using the same stream, and *mode* is set to `SEEK_SET`.

Notes: `fseek` must be implemented by the user.

fsetpos

Syntax:

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

Function: `fsetpos` sets the file position indicator for the stream pointed to by *stream* to the value stored in *pos*. It also clears the end–of–file indicator for *stream*. `fsetpos` returns zero if successful, and returns a nonzero and sets `errno` to a positive value if unsuccessful.

Notes: `fsetpos` must be implemented by the user.

ftell

Syntax:

```
#include <stdio.h>
long int ftell(FILE *stream);
```

Function: `ftell` returns the current file position indicator for the stream pointed to by *stream*. For binary streams this is the number of characters since the beginning of the file. For text streams this is some value that is meaningful to `fseek`, but does not necessarily have any relation to the number of characters. On failure `ftell` returns –1L and stores a positive value in `errno`.

Notes: `ftell` must be implemented by the user.

fwrite

Syntax:

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size,
   size_t n, FILE *stream);
```

Function: `fwrite` writes *n* items of size *size* from a buffer at address *ptr* to the specified output *stream*. `fwrite` returns the number of items written.

fgetc

Syntax:

```
#include <stdio.h>
int getc(FILE *stream);
```

Function: `getc` reads a character from the specified input *stream* and increments the associated file pointer by one byte. `getc` returns the next character from the specified input *stream.*

Notes: If end–of–file is encountered or a read error occurs, `getc` returns EOF.

getchar

Syntax:

```
#include <stdio.h>
int getchar(void);
```

Function: `getchar` is a macro defined as `getc(stdin)`. `getchar` returns the next character from standard input.

getenv

Syntax:

```
#include <stdlib.h>
char *getenv(const char *name);
```

Function: The `getenv` function searches an environment list, provided by the host environment, for the environment variable *name*. It returns the variable definition, or NULL if no definition exists.

Notes: `getenv` must be implemented by the user.

getl

Syntax:

```
#define _EXTENSIONS
#include <stdio.h>
long getl (FILE * stream);
```

Function: `getl` reads and returns a long from the specified file *stream*.

Notes: `getl` is an InterTools extension. The first byte read is the high–order byte of the result.

gets

Syntax:

```
#include <stdio.h>
char *gets(char *s);
```

Function: `gets` reads a string from standard input (stdin) into *s*. The read terminates when a newline character is read or end–of–file is encountered. Any newline character is then discarded and the string is terminated with a null. `gets` returns *s*.

Notes: `fgets` does not discard the newline character. If end–of–file is encountered and no characters have been read, `gets` returns `NULL`.

getw

Syntax:

```
#define _EXTENSIONS
#include <stdio.h>
int getw(FILE *stream);
```

Function: `getw` reads and returns an integer from the specified input *stream.*

Notes: `getw` is an InterTools extension. The first byte read is the high order byte of the result.

gmtime

Syntax:

```
#include <time.h>
struct tm *gmtime(const time_t *time);
```

Function: `gmtime` converts a time value represented by a `time_t` into a time value represented as a `tm` structure, expressed as Coordinated Universal Time (UTC). If UTC is not available, then `gmtime` returns NULL.

Notes: This function must be implemented by the user. The current return value is NULL pointer.

isalnum

Syntax:

```
#include <ctype.h>
int isalnum(int c);
```

Function: `isalnum` tests whether *c* is an alphabetic character (either upper or lower case) or a decimal digit. If *c* is an alphanumeric character, `isalnum` returns TRUE. If *c* is not an alphanumeric character, `isalnum` returns FALSE.

Notes: If *c* is outside the range [–1, 255], the result is undefined.

isalpha

Syntax:

```
#include <ctype.h>
int isalpha(int c);
```

Function: `isalpha` tests whether *c* is an alphabetic character (either upper or lower case). If *c* is a letter, `isalpha` returns TRUE. If *c* is not a letter, `isalpha` returns FALSE.

Notes: If *c* is outside the range [–1, 255], the result is undefined.

iscntrl

Syntax:

```
#include <ctype.h>
int iscntrl(int c);
```

Function: `iscntrl` tests whether *c* is a delete or control character. `iscntrl` returns TRUE if *c* is a control character; FALSE if *c* is not a control character.

Notes: If *c* is outside the range [–1, 255], the result is undefined. The control characters are those whose hex values are between 0 and 1F or equal to 7F.

isdigit

Syntax:

```
#include <ctype.h>
int isdigit(int c);
```

Function: `isdigit` tests whether *c* is a decimal digit. `isdigit` returns TRUE if *c* is a digit; FALSE if *c* is not a digit.

Notes: If *c* is outside the range [–1, 255], the result is undefined.

isgraph

Syntax:

```
#include <ctype.h>
int isgraph(int c);
```

Function: `isgraph` tests whether *c* is a graphic character; that is, any printing character except a space. `isgraph` returns TRUE if *c* is a graphic character. `isgraph` returns FALSE if *c* is not a graphic character.

Notes: If *c* is outside the range [–1, 255], the result is undefined. Graphic characters are those whose hex values are between 21 and 7E. The DEL character is not a graphic character.

islower

Syntax:

```
#include <ctype.h>
int islower(int c);
```

Function: `islower` tests whether *c* is a lowercase alphabetic character. If *c* is a lowercase letter, `islower` returns TRUE. If *c* is not a lowercase letter, `islower` returns FALSE.

Notes: If *c* is outside the range [–1, 255], the result is undefined.

isprint

Syntax:

```
#include <ctype.h>
int isprint(int c);
```

Function: `isprint` tests whether *c* is any printable character, including space. `isprint` returns TRUE if *c* is a printable character; FALSE if *c* is not a printable character.

Notes: If *c* is outside the range [–1, 255], the result is undefined. The printable characters are those whose hex values are between 20 and 7E.

ispunct

Syntax:

```
#include <ctype.h>
int ispunct(int c);
```

Function: `ispunct` tests whether *c* is a punctuation character. Punctuation characters include any printable character except a space, a digit or a letter. If *c* is a punctuation character, `ispunct` returns TRUE. If *c* is not a punctuation character, `ispunct` returns FALSE.

Notes: If *c* is outside the range [–1, 255], the result is undefined. The punctuation characters are those whose hex values are between 21 and 2f, 3A and 40, 5B and 60, or 7B to 7E.

isspace

Syntax:

```
#include <ctype.h>
int isspace(int c);
```

Function: `isspace` tests whether *c* is a white–space character: a space, tab, carriage return, newline, vertical tab or formfeed. If *c* is a white–space character, `isspace` returns TRUE. If *c* is not a white–space character, `isspace` returns FALSE.

Notes: If *c* is outside the range [–1, 255], the result is undefined. The white–space characters are those whose hex values are between 9 and D or equal to 20.

isupper

Syntax:

```
#include <ctype.h>
int isupper(int c);
```

Function: `isupper` tests whether *c* is an upper–case alphabetic character. `isupper` returns TRUE if *c* is an upper–case letter. If *c* is not an upper–case letter, `isupper` returns FALSE.

Notes: If *c* is outside the range [–1, 255], the result is undefined.

isxdigit

Syntax:

```
#include <ctype.h>
int isxdigit(int c);
```

Function: `isxdigit` tests whether *c* is a hexadecimal digit, that is, in the set [0123456789abcdefABCDEF]. `isxdigit` returns TRUE if *c* is a hex digit; FALSE if *c* is not a hex digit.

Notes: If *c* is outside the range [–1, 255], the result is undefined.

labs

Syntax:

```
#include <stdlib.h>
long labs(long x);
```

Function: `labs` calculates |*x*|, the absolute value of the long integer *x*.

Notes: `labs` returns its input if *x* is the most negative long value.

ldexp

Syntax:

```
#include <math.h>
double ldexp(double x, int exp);
```

Function: `ldexp` returns the product of *x* and 2 raised to the integer power *exp*. That is, `ldexp` returns the quantity $x*(2^{exp})$.

Notes: `ldexp` returns IEEE infinity in case of overflow; zero in case of underflow.

ldiv

Syntax:

```
#include <stdlib.h>
ldiv_t ldiv(long int numerator,
   long int denominator);
```

Function: `ldiv` computes the quotient and remainder of the division of the *numerator* by the *denominator*. The type *ldiv_t* is a structure that contains two components, `quot` and `rem`, both of type `long int`.

Notes: If the result cannot be represented the behavior is undefined. Otherwise `quot` * *denominator* + `rem` shall equal *numerator*.

localeconv

Syntax:

```
#include <locale.h>
struct lconv *localeconv(void);
```

Function: `localeconv` returns a pointer to a filled–in structure that contains numeric formatting information for the current locale. The values in the structure cannot be changed by the program except by later calls to `localeconv` or on calls to `setlocale` that change the categories `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC`.

Notes: The locale is the ANSI C method of specifying culturally–dependent information. Currently the library supports only the default or "C" locale.

localtime

Syntax:

```
#include <time.h>
struct tm *localtime(const time_t *time);
```

Function: `localtime` breaks down a time value expressed as a `time_t` into a time value expressed as a `tm` structure.

Notes: The `tm` structure pointed to by the return value may be overwritten by subsequent calls to `localtime` or `gmtime`.

log

Syntax:

```
#include <math.h>
double log(double x);
```

Function: `log` computes the natural logarithm of *x*. If *x* is zero, then `log` returns IEEE negative infinity. If *x* is less than zero, `log` returns IEEE NaN (Not a Number) and sets `errno` to EDOM.

log2

Syntax:

```
#define _EXTENSIONS
#include <math.h>
double log2(double x);
```

Function: `log2` computes the base 2 logarithm of the double *x*. If *x* is zero then `log2` returns IEEE negative infinity. If *x* is less than zero, `log2` returns IEEE NaN (Not a Number) and sets `errno` to EDOM.

Notes: `log2` is an InterTools extension.

log10

Syntax:

```
#include <math.h>
double log10(double x);
```

Function: `log10` computes the base 10 logarithm of *x*. If *x* is zero, then `log10` returns IEEE negative infinity. If *x* is less than zero, `log10` returns IEEE NaN (Not a Number) and sets `errno` to EDOM.

longjmp

Syntax:

```
#include <setjmp.h>
void longjmp(jmp_buf x, int n);
```

Function: `longjmp` returns to an environment established by `setjmp` using *n* as the return value, except that 1 is returned if *n* is zero. If `setjmp` was not invoked with this environment or if the function containing the invocation has terminated, then the results are undefined.

The values of non–volatile objects of automatic storage class (that is, non–volatile local variables) local to the function calling `setjmp` are indeterminate if they were modified between the `setjmp` and `longjmp` calls.

After `longjmp` is completed, program execution continues as if the corresponding invocation of `setjmp` had just returned *n* (or 1, if *n* is zero).

malloc

Syntax:

```
#include <stdlib.h>
void *malloc(size_t nwords);
```

Function: `malloc` allocates space of the size *nwords* on the heap. Allocated memory is not initialized to zero.

Notes: If memory space is exhausted, `malloc` returns a null pointer.

mblen

Syntax:

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

Function: If *s* is a null pointer or points to a null character, then `mblen` returns 0. Otherwise, `mblen` returns the number of bytes comprising the multi–byte character *s*. If *s* points to an invalid multibyte character, then `mblen` returns –1.

`mblen` assumes the Shift JIS convention for Japanese character encoding. Values between 1 and 127 (hex 7F) are treated as one–byte ASCII codes. Values between 160 and 223 (hex A0 to DF) are treated as one–byte kana codes. Kanji characters are encoded as two–byte sequences where the first byte is between 129 and 159 (hex 81 to 9F) or 224 to 252 (hex E0 to FC) and the second byte is between 64 and 252 (hex 40 to FC).

Here is a summary:

ASCII (one byte)	0 through 0x7F
Kana (one byte)	0xA0 through 0xDF
Kanji (two bytes)	first byte: 0x81 through 0x9F and 0xE0 through 0xFC second byte: 0x40 thorugh 0xFC

mbstowcs

Syntax:

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char
*s,
    size_t n);
```

Function: `mbstowcs` converts a sequence of multi–byte characters pointed to by *s* to wide characters, and stores no more than *n* of them in the array of wide characters pointed to by *pwcs*. It copies up to to *n* characters from *s* to *pwcs*, until it reaches a null character. If an invalid character is found, `mbstowcs` returns –1. Otherwise, the number of characters written is returned. The `mbstowcs` routine assumes the Shift JIS convention for Japanese characters. See `mblen` for more details.

mbstowc

Syntax:

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s,
   size_t n);
```

Function: `mbtowc` converts the multi–byte character pointed to by *s* to a wide character, and stores the wide character in the location pointed to by *pwc*. It returns 0 if either *s* is null or points to the null character and returns –1 if the character is invalid. Otherwise, if *pwc* is not null, the character at *s* is stored at *pwc* and `mbtowc` returns 1 if the character is a one–byte ASCII or kana code and returns 2 if it is a two–byte Kanji code. The `mbstowcs` routine assumes the Shift JIS convention for Japanese characters. See `mblen` for more details.

memccpy

Syntax:

```
#include <extended.h>
void *memccpy(void *s1, void *s2, char c,
   size_t n);
```

Function: memccpy copies characters from *s2* to *s1*, stopping after the first occurrence of character *c* has been copied, or after *n* characters have been copied, whichever comes first. memccpy returns a pointer to the character after the copy of *c* in *s1*, or NULL if *c* was not encountered.

Notes: memccpy is an InterTools extension.

memchr

Syntax:

```
#include <string.h>
void *memchr(const void *s, int c,
   size_t n);
```

Function: `memchr` searches an *n* word memory area at address *s* for the character *c*. `memchr` returns a pointer to the first occurrence of *c*. If *c* is not found, `memchr` returns NULL.

Notes: `memchr` is equivalent to `strchr`, except it does not stop at nulls.

memcmp

Syntax:

```
#include <string.h>
int memcmp(const void *s1, const void *s2,
   size_t n);
```

Function: memcmp compares *n* words of memory at addresses *s1* and *s2*. memcmp returns zero if the memory areas are equal, an integer greater than zero if *s1* is lexically larger than *s2*; else memcmp returns an integer less than zero.

Notes: memcmp is equivalent to the strcmp routine, except it does not stop at nulls.

memcpy

Syntax:

```
#include <string.h>
void *memcpy(void *s1, const void *s2,
   size_t n);
```

Function: memcpy copies *n* words of memory from *s2* to *s1*. memcpy returns its first argument, *s1*.

Notes: memcpy is equivalent to strncpy except that it does not stop at nulls. If the two memory areas overlap then the results are undefined. See also memmove.

memmove

Syntax:

```
#include <string.h>
void *memmove(void *s1, const void *s2,
   size_t n);
```

Function: memmove copies *n* words of memory from *s2* to *s1*. If the two memory areas overlap, then the copy is done as if the characters are first copied from *s2* into a temporary area of size *n*, and then copied to *s1*. memmove returns its first argument, *s1*.

memset

Syntax:

```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

Function: `memset` fills *n* words of memory at *s* with a fill character *c* and returns its first argument, *s*.

mktime

Syntax:

```
#include <time.h>
time_t mktime(struct tm *time);
```

Function: `mktime` converts a time value, expressed as a `tm` structure, into a time value expressed as a `time_t`. The original values of the `tm_wday` and `tm_yday` components of the `tm` structure are ignored, and the other values are not restricted to their usual ranges.

On successful completion the values of the `tm_wday` and `tm_yday` components of the `tm` structure are set appropriately, and the other values are normalized to be in their usual ranges.

Notes: Some time values which can be expressed as `tm` structures cannot be expressed as `time_t` values. In that case `(time_t)` –1 is returned.

modf

Syntax:

```
#include <math.h>
double modf(double x, double *intptr);
```

Function: `modf` returns the fractional part of *x* and stores the integral part indirectly through the pointer *intptr*. In effect, this breaks the double *x* into an integer and fractional part. The breakdown into integer and fractional part is defined by truncation (round towards zero).

For example, the integer part of –3.9 is –3.0, and the fractional part is –.9.

offsetof

Syntax:

```
#include <stddef.h>
size_t offsetof(type, member);
```

Function: The macro `offsetof` returns the offset in bytes from the beginning of the structure *type* to the structure member *member*.

Notes: The offset macro is defined as follows:

```
#define  offsetof(type, member)
         (size_t) & (((type *)0) -> member)
```

perror

Syntax:

```
#include <stdio.h>
void perror(const char *s);
```

Function: `perror` maps the error number in the integer expression *errno* to an error message. It writes a sequence of characters to the standard error stream as follows. First, if *s* is not a null pointer and does not point to a null character, the string pointed to by *s* is printed, followed by a colon and a space. Next, the appropriate error string is printed, followed by a newline.

See Also: strerror.

pow

Syntax:

```
#include <math.h>
double pow(double x, double y);
```

Function: pow returns *x* raised to the *y* power.

If *x* is zero and *y* is non–positive, then IEEE infinity is returned and errno is set to EDOM.

If *x* is negative and *y* is not an integer, then IEEE infinity is returned and errno is set to EDOM.

Notes: To avoid errors, *x* must be greater than or equal to zero, unless *y* is an integer.

printf

Syntax:

```
#include <stdio.h>
int printf(const char *format, ...);
```

Function: `printf` converts and formats its arguments and prints them to stdout, following specifications of *format*. *format* may contain ordinary characters, which are simply copied to standard output, and conversion specifications, each of which causes conversion and printing of the next successive argument to `printf`.

The following description is taken from the ANSI C standard.

Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

- Zero or more flags (in any order) that modify the meaning of the conversion specification.
- An optional minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces (by default) to the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk * (described later) or a decimal integer. Note that zero (0) is taken as a flag, not as the beginning of a field width.

- An optional precision that gives the minimum number of digits to appear for the d, i, o, u, x and X conversion, the number of digits to appear after the decimal–point character for e, E and f conversions, the maximum number of significant digits for the g and G conversions, or the maximum number of characters to be written from a string in s conversion. The precision takes the form of a period (.) followed either by an asterisk * (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.

- An optional h specifying that a following d, i, o, u, x or X conversion specifier applies to a short int or unsigned short int argument (the argument will have been promoted according to integral promotions, and its value shall be converted to short int or unsigned short int before printing) an optional h specifying that a following n conversion specifer applies to a pointer to a short int argument; an optional l specifying that a following d, i, o, u, x or X conversion specifier applies to a long int or unsigned long int argument; an optional l specifying that a following n conversion specifier applies to a pointer to a long int argument; or an optional L specifying that a following e, E, f, g or G conversion specifier applies to a long double argument. If an h, l or L appears with any other conversion specifier, the behavior is undefined.

- A character that specifies the type of conversion to be applied.

As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an `int` argument supplies the field width or precision. The arguments specifying field width, or precision or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a – flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

The flag characters and their meanings are:

– The result of the conversion will be left–justified within the field. (It will be right–justified if this flag is not specified.)

\+ The result of a signed conversion will always begin with a plus or minus sign. (It will begin with a sign only when a negative value is converted if this flag is not specified).

space
If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space will be prefixed to the result. If the space and + flags both appear, the space flag will be ignored.

\# The result is to be converted to an "alternate form." For `o` conversion, it increases the precision to force the first digit of the result to be a zero. For `x` (or `X`) conversion, a nonzero result will have `0x` (or `0X`) prefixed to it. For `e`, `E`, `f`, `g` and `G` conversions, the result will always contain a decimal–point character, even if no digits follow it. (Normally, a

decimal–pointer character appears in the result of these conversions only if a digit follows it.) For `g` and `G` conversions, trailing zeros will *not* be removed from the result. For other conversions, the behavior is undefined.

`0` For `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g` and `G` conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the `0` and – flags both appear, the `0` flag will be ignored. For `d`, `i`, `o`, `u`, `x` and `X` conversions, if a precision is specified, the `0` flag will be ignored. For other conversions, the behavior is undefined.

The conversion specifiers and their meanings are:

`d, i`
: The `int` argument is converted to signed decimal in the style *[–]dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

`o, u, x, X`
: The `unsigned int` argument is converted to unsigned octal (`o`), unsigned decimal (`u`) or unsigned hexadecimal notation (`x` or `X`) in the style *dddd*; the letters `abcdef` are used for `x` conversion and the letters `ABCDEF` for `X` conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

`f`
: The `double` argument is converted to decimal notation in the style *[–]ddd.ddd*, where the number of digits after the decimal–point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal–point character appears. If a decimal–point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

e, E
The `double` argument is converted in the style *[–]d.ddde±dd*, where there is one digit before the decimal–point character (which is nonzero if the argument is nonzero) and the number of digits after it is equal to the precision; if the precision is missing it is taken as 6; if the precision is zero and the # flag is not specified, no decimal point character appears. The value is rounded to the appropriate number of digits. The `E` conversion specifier will produce a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits. If the value is zero, the exponent is zero.

g, G
The `double` argument is converted in style `f` or `e` (or in style `E` in the case of a `G` conversion specifier) with the precision specifying the number of significant digits. If the precision is zero, it is taken as 1. The style used depends on the value converted; style `e` (or `E`) will be used only if the exponent resulting from such a conversion is less than –4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a decimal–point character appears only if it is followed by a digit.

c The `int` argument is converted to an `unsigned char`, and the resulting character is written.

s The argument shall be a pointer to an array of character type. (No special provisions are made for multibyte characters.) Characters from the array are written up to (but not including) a terminating null character. If the precision is specified, no more than

that many characters are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character.

`p` The argument shall be a pointer to `void`. The value of the pointer is converted to a sequence of hex characters, just like the `x` format.

`n` The argument shall be a pointer to an integer into which is *written* the number of characters written to the output stream so far by this call to `printf`. No argument is converted.

`%` A % is written. No argument is converted. The complete conversion specification shall be %%.

If a conversion specification is invalid, the behavior is undefined.

If any argument is, or points to, a union or an aggregate (except for an array of character type using `%s` conversion, or a pointer using `%p` conversion), the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

Notes: `printf` uses its first argument to decide how many arguments follow and what their types are. `printf` normally returns the number of characters printed. If there are not enough arguments, or arguments are of the wrong type, the results are undefined.

`printf` returns the number of characters printed, or a negative number if an output error occurred. No more than 509 characters shall be produced for any single conversion.

`fprintf` and `sprintf` are identical to `printf`, except that `fprintf` directs its output to the specified output *stream*; `sprintf` directs its output to the string *s*.

putc

Syntax:

```
#include <stdio.h>
int putc(int c, FILE *stream);
```

Function: `putc` writes the character *c* to the specified output *stream*.

putchar

Syntax:

```
#include <stdio.h>
int putchar(int c);
```

Function: `putchar` is equivalent to `putc` with the second argument of *stdout*. `putchar` returns the character written. If an error occurs, `putchar` returns EOF.

putl

Syntax:

```
#define _EXTENSIONS
#include <stdio.h>
long putl(long l, FILE *stream);
```

Function: `putl` writes a long, *l*, to the specified output *stream* and returns its first argument.

Notes: `putl` is equivalent to `putw` if integers are 32 bits. The first character written is the high–order byte of *l*.

`putl` is an InterTools extension.

puts

Syntax:

```
#include <stdio.h>
int puts(const char *s);
```

Function: `puts` copies the string *s* to standard output (`stdout`). `puts` returns 0, the success code.

putw

Syntax:

```
#define _EXTENSIONS
#include <stdio.h>
int putw(int w, FILE *stream);
```

Function: `putw` writes an integer *w* to the specified output *stream* and returns its first argument.

Notes: The first character written is the high–order byte of w.

qsort

Syntax:

```
#include <stdlib.h>
void qsort(void *base, size_t
    nummembr, size_t size,
    int (*comp)(const void *,
    const void *));
```

Function: `qsort` sorts an array of *nummembr* elements, the first element of which is pointed to by *base*. The size of each object is specified by *size*. The contents of the array are sorted into ascending order according to the comparison function pointed to by *comp*, which takes two arguments that point to the objects being compared. The comparison function should return an integer less than, equal to, or greater than zero if the first argument is considered to be less than, equal to, or greater than the second.

raise

Syntax:

```
#include <signal.h>
int raise(int sig);
```

Function: The `raise` function sends the signal *sig* to the executing program. `raise` returns zero if successful, and nonzero if unsuccessful.

Notes: See `signal` for more details.

rand

Syntax:

```
#include <stdlib.h>
int rand(void);
```

Function: `rand` computes and returns pseudo–random integers in the range [0, 32767]. The pseudo–random number is computed via a simple multiplicative congruence algorithm based on a "seed" value (initially one). At any time the seed value may be reset using the `srand` routine.

Notes: The low bits of the numbers generated are not very random; use the middle bits. Specifically, the lowest bit alternates between 0 and 1.

rcopy

Syntax:

```
#include <rcopy.h>
void rcopy(struct hdr *addr)
```

Function: `rcopy` is a ROM initialization utility, called at the start of a new C program. Its argument *addr* is the address of an initialization segment created by the `llink` utility.

Notes: `rcopy` should be called at the start of a new program or during system restart. The argument to `rcopy` is typically the address of a fictitious external variable. The name of this variable is carefully chosen to match that generated by `llink` for the initialization segment itself. The name of the initialization segment can be determined by `llink` options. Note that the C compiler prepends an underscore to the names of external variables, so an external variable named `x` would match with an initialization segment named `_x`.

`rcopy` should be called once for each initialization segment created by `llink`. Refer to the *Linking Locator* chapter in the *InterTools User's Manual* for more details.

`rcopy` is an InterTools extension.

realloc

Syntax:

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

Function: `realloc` changes the size of the object pointed to by *ptr* to the size specified by *size*. The contents of the object will remain unchanged up to the lesser of the new and old sizes. If *ptr* is a null pointer, `realloc` behaves like the `malloc` function for the specified size. If *size* is zero and *ptr* is not a null pointer, the object is freed.

remove

Syntax:

```
#include <stdio.h>
int remove(const char *filename);
```

Function: `remove` deletes *filename* and returns zero upon successful completion.

Notes: `remove` must be implemented by the user.

rename

Syntax:

```
#include <stdio.h>
int rename(const char *old,
   const char *new);
```

Function: This routine renames the file *old* to the filename *new* and returns zero if successful.

Notes: `rename` must be implemented by the user.

rewind

Syntax:

```
#include <stdio.h>
void rewind(FILE *stream);
```

Function: The `rewind` function sets the file position indicator for the stream pointed to by `stream` to the beginning of the file. It is equivalent to:

```
(void) fseek(stream, 0L, SEEK_SET)
```

except that the error indicator for the steam is also cleared.

Notes: The `rewind` function returns no value.

`rewind` must be implemented by the user.

scanf

Syntax:

```
#include <stdio.h>
int scanf(const char *format, ...);
```

Function: `scanf` reads input from standard input under control of the string pointed to by *format*. The format string specifies admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted values. If there are insufficient arguments or the types of the arguments do not match the converted values the behavior is undefined. `scanf` returns EOF if an input failure occurs before any conversion. Otherwise `scanf` returns the number of items assigned.

The following description of the format specification is taken from the ANSI C standard.

The format is composed of zero or more directives: one or more white–space characters; an ordinary multibyte character (neither % nor a white–space character); or a conversion specification. Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

- An optional assignment–suppressing character `*`.
- An optional nonzero decimal integer that specifies the maximum field width.
- An optional `h`, `l` or `L` indicating the size of the receiving object. The conversion specifiers `d`, `i`, and `n` shall be preceded by `h` if the corresponding

argument is a pointer to `short int` rather than a pointer to `int`, or by `l` if it is a pointer to `long int`. Similarly, the conversion specifiers `o`, `u`, and `x` shall be preceded by `h` if the corresponding argument is a pointer to `unsigned short int` rather than a pointer to `unsigned int`, or by `l` if it is a pointer to `unsigned long int`. Finally, the conversion specifiers `e`, `f`, and `g` shall be preceded by `l` if the corresponding argument is a pointer to `double` rather than a pointer to `float`, or by `L` if it is a pointer to `long double`. If an `h`, `l`, or `L` appears with any other conversion specifier, the behavior is undefined.

- A character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

The `scanf` function executes each directive of the format in turn. If a directive fails, as detailed below, the `scanf` function returns. Failures are described as input failures (due to the unavailability of input characters) or matching failures (due to inappropriate input).

A directive composed of white–space character(s) is executed by reading input up to the first non–white–space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary multibyte character is executed by reading the next character of the stream. If one of the characters differs from one comprising the directive, the directive fails, and the differing and subsequent characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

Input white–space characters (as specified by the `isspace` function) are skipped, unless the specification includes a `[`, `c`, or `n` specifier. (These white–space characters are not counted against a specified field width.)

An input item is read from the stream, unless the specification includes an `n` specifier. An input item is defined as the longest matching sequence of input characters, unless that exceeds a specified field width, in which case it is the initial subsequence of that length in the sequence. The first character, if any, after the input

item remains unread. If the length of the input item is zero, the execution of the directive fails: this condition is a matching failure, unless an error prevented input from the stream, in which case it is an input failure.

Except in the case of a `%` specifier, the input item (or, in the case of the `%n` directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a `*`, the result of the conversion is placed in the object pointed to by the first argument following the *format* argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The following conversion specifiers are valid:

`d` Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the `strtol` function with the value 10 for the *base* argument. The corresponding argument shall be a pointer to integer.

`i` Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the `strtol` function with the value 0 for the *base* argument. The corresponding argument shall be a pointer to integer.

`o` Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 8

for the *base* argument. The corresponding argument shall be a pointer to unsigned integer.

`u` Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 10 for the *base* argument. The corresponding argument shall be a pointer to unsigned integer.

`x` Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 16 for the *base* argument. The corresponding argument shall be a pointer to unsigned integer.

`e, f, g`
Matches an optionally signed floating–point number, whose format is the same as expected for the subject string of the `strtod` function. The corresponding argument shall be a pointer to floating.

`s` Matches a sequence of non–white–space characters. The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically.

`[` Matches a nonempty sequence of characters from a set of expected characters (the *scanset*). The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically. The conversion specifier includes all subsequence characters in the *format* string, up to and including the matching right bracket (`]`). The characters between the brackets (the *scanlist*) comprise the scanset, unless the character after the left bracket is a circumflex(^), in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with `[]` or `[^]`, the right bracket character is in the scanlist and the next right bracket character is the matching right bracket that ends the specification; otherwise the first right bracket character is the one that ends the specification.

`c` Matches a sequence of characters of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence. No null character is added. No special provisions are made for multibyte characters.

`p` Matches a sequence of hexadecimal characters whose format is the same as expected for the subject sequence of the `strtoul` function with the value 16 for the base argument. The corresponding argument shall be a pointer to a pointer to `void`. This matches the `%p` conversion by `printf`.

`n` No input is consumed. The corresponding argument

shall be a pointer to integer into which is to be written the number of characters read from the input stream so far by this call to the `scanf` function. Execution of a `%n` directive does not increment the assignment count returned at the completion of execution of the `scanf` function.

`%` Matches a single `%`; no conversion or assignment occurs. The complete conversion specification shall be `%%`.

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers `E`, `G` and `X` are also valid and behave the same as, respectively, `e`, `g` and `x`.

If end–of–file is encountered during input, conversion is terminated. If end–of–file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white–space (including new–line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the `%n` directive.

Notes: The `scanf` function returns the value of the macro `EOF` if an input failure occurs before any conversion. Otherwise, the `scanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

`scanf` is identical to the `fscanf` and `sscanf` routines, except `fscanf` reads its input from the specified input *stream*; `sscanf` reads its input from the string *s*.

setbuf

Syntax:

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

Function: `setbuf` may be used only after the stream pointed to by *stream* has been opened but before any other operation is performed on the stream. If *buf* is NULL, then it causes the stream to be unbuffered. Otherwise *buf* must point to an array of size `BUFSIZE` and *buf* is used as the buffer in a fully buffered file.

Notes: `setbuf` must be implemented by the user.

setjmp

Syntax:

```
#include <setjmp.h>
int setjmp(jum_buf x);
```

Function: `setjmp` establishes an environment for later use by `longjmp`. The type of the argument, `jmp_buf`, is provided in the `setjmp.h` include file.

Notes: See `longjmp` for more details.

setlocale

Syntax:

```
#include <locale.h>
char *setlocale(int category,
   const char *locale);
```

Function: `setlocale` returns a pointer to a string that describes the new *locale*, or a null pointer if *locale* cannot be changed. The value of *category* must match the value of one of the macros defined in the header file `locale.h`. These macros begin with `LC_`.

If *locale* is a null pointer, `setlocale` does not change the current *locale*. If *locale* points to the string "C", the new *locale* is the "C" *locale* for the category specified. If *locale* points to the string " ", the *locale* is the native *locale* for the category specified. *locale* can point to strings returned from previous calls to `setlocale`.

At program startup, the target environment acts as if the call `setlocale(`*"LC_ALL","C"*`)` was called before it called *main*.

Notes: The supplied `setlocale` only works for the "C" locale.

setvbuf

Syntax:

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf,
   int mode, size_t size);
```

Function: `setvbuf` sets the buffering mode for the stream pointed to by *stream*, according to *buf*, *mode* and *size*. If *buf* is not a null pointer, then *buf* is the address of the first element of an array of chars of size *size* that can be used as a stream buffer. *mode* must be one of the following macros: _IOFBF (full buffering), _IOLBF (line buffering) _IONBF (no buffering).

`setvbuf` must be called immediately after a call to `fopen` to associate a file with that stream.

Notes: `setvbuf` must be implemented by the user.

signal

Syntax:

```
#include <signal.h>
void (*signal(int sig,
   void (*func)(int)))(int);
```

Function: The `signal` function allows a program to specify what action shall be taken upon receipt of the signal *sig*, which may be generated externally, or may be explicitly generated with the `raise` function. If the value of *func* is `SIG_DFL`, default handling for that signal will occur. If the value of *func* is `SIG_IGN`, the signal will be ignored. Otherwise, the signal handler pointed to by *func* will be called upon receipt of signal *sig*. If no errors are detected, `signal` returns the value of *func* for the most recent call to `signal` for the specified flag *sig*. Otherwise, a value of `SIG_ERR` is returned and a positive value is stored in `errno`.

Notes: Since most embedded applications will not want to abort when errors are detected, the library as distributed does not raise signals under any circumstances.

sin

Syntax:

```
#include <math.h>
double sin(double x);
```

Function: `sin` computes the sine of *x*, expressed in radians.

sinh

Syntax:

```
#include <math.h>
double sinh(double x);
```

Function: `sinh` computes the hyperbolic sine of *x*.

sprintf

Syntax:

```
#include <stdio.h>
int sprintf(char *s, const char *format,
    ...);
```

Function: See `printf`.

Notes: `sprintf` is identical to `printf`, but directs its output to the string *s*. See `printf` for more details.

sqrt

Syntax:

```
#include <math.h>
double sqrt(double x);
```

Function: `sqrt` computes the square root of *x*. If *x* is less than zero, `errno` is set to EDOM and an IEEE NaN (Not a Number) is returned.

srand

Syntax:

```
#include <stdlib.h>
void srand(unsigned int seed)
```

Function: `srand` resets the seed for the random number generator `rand` with the value *seed*.

sscanf

Syntax:

```
#include <stdio.h>
int sscanf(const char *s,
   const char *format, ...);
```

Function: See `scanf`.

Notes: `sscanf` is identical to `fscanf` and `scanf`, but reads its input from the string *s*. See `scanf` for more details.

strcat

Syntax:

```
#include <string.h>
char *strcat(char *s1, const char *s2);
```

Function: `strcat` appends a copy of the string pointed to by *s2* (including the terminating null character) to the end of the string pointed to by *s1*. The initial character of *s2* overwrites the null characters at the end of *s1*.

Notes: *s1* must contain enough room to hold the resulting string.

strchr

Syntax:

```
#include <string.h>
char *strchr(const char *s, int c);
```

Function: `strchr` looks for the first occurrence of a specific character, *c*, in a null terminated target string, *s*. `strchr` returns a pointer to the first character that matches *c*. If no character matches *c*, `strchr` returns NULL.

Notes: In ANSI C, `strchr` replaces the Unix `index` routine.

strcmp

Syntax:

```
#include <string.h>
int strcmp(char *s1, const char *s2);
```

Function: `strcmp` compares the strings *s1* and *s2*, character by character, for lexical order in the character collating sequence. `strcmp` returns zero if the strings are equal; an integer greater than zero if *s1* is lexically larger than *s2*. If *s1* is lexically smaller than *s2*, `strcmp` returns an integer less than zero.

strcoll

Syntax:

```
#include <string.h>
int strcoll(const char *s1, const char
*s2);
```

Function: `strcoll` compares the string *s1* to the string *s2*, and with both strings interpreted as appropriate to the `LC_COLLATE` setting of the current locale. The routine returns an integer greater than, equal to, or less than zero, if *s1* is greater than, equal to, or less than *s2*, interpreted according to the locale.

Notes: In the "C" locale, `strcoll` is equivalent to `strcmp`.

strcpy

Syntax:

```
#include <string.h>
char *strcpy(char *s1, const char *s2);
```

Function: `strcpy` copies the string *s2* and its terminating null to *s1* and returns *s1*.

Notes: *s1* must contain enough room to hold the result.

strcspn

Syntax:

```
#include <string.h>
size_t strcspn(char *s1, const char *s2);
```

Function: `strcspn` scans the string starting at *s1* for the first occurrence of a character in the string starting at *s2*. `strcspn` returns the length of the initial segment of *s1*, which consists entirely of characters not in *s2*.

strerror

Syntax:

```
#include <string.h>
char *strerror(int errnum);
```

Function: `strerror` maps the error number in *errnum* to an error message which `strerror` returns. The message buffer is static and is overwritten by subsequent calls to `strerror`.

strftime

Syntax:

```
#include <time.h>
size_t strftime(char *s, size_t maxsize,
   const char *format,
   const struct tm *timeptr);
```

Function: `strftime` puts characters into the array, *s*, as specified by the format string, *format*. The format string consists of zero or more conversion specifiers and ordinary characters. The conversion specifiers are given below. Any ordinary characters including the terminating null character are copied unchanged into the array. No more than *maxsize* characters are placed into the array. The values used in the conversions are contained in the structure *timeptr*.

If the total number of resulting characters is not more that *maxsize*, then `strftime` returns the number of characters copied into the array, *s*, otherwise `strftime` puts *maxsize* characters in the array *s* and returns the value zero.

Format specifiers:

`%a` is replaced by the locale's abbreviated weekday name.

`%A` is replaced by the locale's full weekday name.

`%b` is replaced by the locale's abbreviated month name.

`%B` is replaced by the locale's full month name.

`%c` is replaced by the locale's appropriate date and time

representation.

`%d` is replaced by the day of the month (01–31).

`%H` is replaced by the hour of the day, 24 hour clock (00–23).

`%I` is replaced by the hour of the day, 12 hour clock (01–12).

`%j` is replaced by the day of the year (001–366).

`%m` is replaced by the month (01–12).

`%M` is replaced by the minute (00–59).

`%P` is replaced by the locale's AM/PM designation associated with a 12 hour clock.

`%S` is replaced by the seconds (00–59).

`%U` is replaced by the week number of the year, with the first Sunday of the year as the first day of week 1 (00–53).

`%w` is replaced by the weekday, Sunday = 0, Saturday = 6.

`%W` is replaced by the week number of the year, with the first Monday of the year as the first day of week 1 (00–53).

`%x` is replaced by the date representation.

`%X` is replaced by the time representation.

`%y` is replaced by the year without the century (00–99).

`%Y` is replaced by the year with the century.

`%Z` is replaced by no characters.

`%%` is replaced by `%`.

strlen

Syntax:

```
#include <string.h>
size_t strlen(const char *s);
```

Function: `strlen` scans the text string starting at *s* and returns the number of characters it encounters before the first null character.

strncat

Syntax:

```
#include <string.h>
char *strncat(char *s1, const char *s2, size_t n);
```

Function: `strncat` appends up to *n* characters from the string *s2* to the end of string *s1*, and then terminates the string with a null. `strncat` returns its first argument, *s1*.

Notes: `strncat` is identical to `strcat`, except it appends a limit of *n* characters, plus one for the null.

strncmp

Syntax:

```
#include <string.h>
char *strncmp(char *s1,
   const char *s2, size_t n);
```

Function: `strncmp` compares two text strings, *s1* and *s2*, character by character, for lexical order in the character collating sequence. The first string starts at *s1*, the second at *s2*. *n* specifies the maximum number of characters to be compared, unless the terminating null in *s1* or *s2* is encountered first. The strings must match, including any terminating null characters that may be encountered, in order for them to be equal. `strncmp` returns an integer greater than zero if *s1* is lexically greater than *s2*, zero if *s1* is lexically equal to *s2*, and an integer less than zero if *s1* is lexically less than *s2*.

Notes: `strncmp` is identical to `strcmp`, except it compares a maximum of *n* characters.

strncpy

Syntax:

```
#include <string.h>
char *strncpy(char *s1,
   const char *s2, size_t n);
```

Function: `strncpy` copies characters from *s2* to *s1* until it reaches the end of *s2* or until *n* characters have been copied. `strncpy` pads with zeros, if necessary, to copy *n* characters total. `strncpy` returns its first argument, *s1*.

Notes: If the string *s2* is longer than *n* characters, *s1* may not end with a null character.

strpbrk

Syntax:

```
#include <string.h>
char *strpbrk(const char *s1,
   const char *s2);
```

Function: `strpbrk` scans the string *s1* for the first occurrence of a character in the string *s2*. `strpbrk` returns a pointer to the first character in *s1* that is also in *s2*, or null if *s1* has no characters from *s2*.

strrchr

Syntax:

```
#include <string.h>
char *strrchr(const char *s, int c);
```

Function: `strrchr` looks for the last occurrence of a specific character, *c*, in a null terminated target string, *s*. `strrchr` returns a pointer to the last character that matches *c*. If no character matches *c*, `strrchr` returns NULL.

Notes: In ANSI C, `strrchr` replaces the Unix `rindex` routine.

strspn

Syntax:

```
#include <string.h>
size_t strspn(char *s1, const char *s2);
```

Function: `strspn` scans the string *s1* for the first occurrence of a character not in the string *s2*. `strspn` returns the length of the initial segment of *s1* which consists entirely of characters in *s2*.

strstr

Syntax:

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

Function: `strstr` finds the first instance of the string *s2* in the string *s1*. `strstr` returns a pointer to the occurrence of *s1* if found or a null pointer if the string was not found. If *s2* points to a string of zero length, `strstr` returns *s1*.

strtod

Syntax:

```
#include <stdlib.h>
double strtod(const char *s, char **endptr);
```

Function: `strtod` converts a string *s*, into a double floating point type. `strtod` is identical to `atof`, except that it stores a pointer to the remainder of the string in the object pointed to by *endptr*, providing that *endptr* is not NULL. Leading white–space characters (as defined by the `isspace` function) are allowed.

If `strtod` detects a format error in the string, it returns zero. If the correct value is outside the range of representable values, plus or minus `HUGE_VAL` is returned and the value of the macro `ERANGE` is stored in `errno`.

strtok

Syntax:

```
#include <string.h>
char *strtok(char *s1, const char *s2);
```

Function: `strtok` breaks a string into tokens. Consider *s1* as a sequence of zero or more tokens separated by spans of one of more characters from the "separator" string, *s2*. The first call to `strtok` returns a pointer to the first token in *s1*, and will have a null written at the end of the token. The function keeps track of its position in *s1*, and subsequent calls work through *s1* after the last token returned. When no tokens remain, `strtok` returns NULL. The *s2* string may be different from call to call.

Notes: This routine will not operate correctly if *s1* points to ROM (read–only memory).

strtol

Syntax:

```
#include <stdlib.h>
long strtol(const char *s,
   char **endptr, int base);
```

Function: `strtol` converts a string, *s*, into a long integer. The string that will be converted is the longest string which matches the following pattern:

```
optional white-space
optional +/- sign,
optional 0x or 0X,
0-9, a-z, A-Z
```

Here *a–z* represent 10 to 35 for bases greater than 10. The 0x or 0X are only allowed when the base is zero or 16. `strtol` only understands letters that are less than the base. If base equals 0, and there is a leading 0x or 0X, the base is assumed to be 16. If there is a leading 0, the base is octal (8). Any other initial patterns are considered to be decimal (base 10) numbers.

**endptr* is set to the remainder of the string that was not converted to a long.

If *s* does not contain a valid pattern, then **endptr* is set equal to *s*, and the `strtol` returns zero. If the value is too large to be represented by a long, `strtol` sets `errno` to ERANGE, returns `LONG_MAX` for positive numbers, and returns `LONG_MIN` for negative numbers.

strtoul

Syntax:

```
#include <stdlib.h>
unsigned long strtoul(const char *s,
   char **endptr, int base);
```

Function: `strtoul` converts a string, *s*, into an integer of type unsigned long integer. `strtoul` is identical to `strtol`, except that it converts the string to an unsigned long int.

**endptr* is set to the remainder of the string that was not converted to a long.

If *s* does not contain a valid pattern, **endptr* is set equal to *s*, and the `strtoul` returns zero. If the value is too large to be represented by an unsigned long, `strtoul` sets `errno` to ERANGE, and returns ULONG_MAX.

strxfrm

Syntax:

```
#include <string.h>
size_t strxfrm(char *s1, const
   char *s2, size_t n);
```

Function: `strxfrm` transforms the input string pointed to by *s2* and places the result in *s1*. No more than *n* characters are stored in *s1*, including the terminating null character. The transformation is such that `strcmp` on the transformed strings yields the same value as `strcoll` on the untransformed strings.

If *n* is zero, then *s1* may be null. `strxfrm` returns the length of the transformed string, not including the null character. If the value returned is *n* or more, then no characters are copied.

swab

Syntax:

```
#include <extended.h>
void swab(char *from, char *to, int nbytes);
```

Function: `swab` copies *nbytes* bytes pointed to by *from* to the position pointed to by *to*, exchanging adjacent even and odd bytes.

Notes: *nbytes* should be an even number.

If *from* and *to* data areas overlap, the results are undefined.

`swab` is an InterTools extension.

system

Syntax:

```
#include <stdlib.h>
int system(const char *str);
```

Function: `system` passes the string *str* to the host environment's command processor. If *str* is a null pointer, `system` can be used to inquire whether a command processor exists, in which case `system` returns nonzero only if a command processor exists.

Notes: `system` must be implemented by the user.

tan

Syntax:

```
#include <math.h>
double tan(double x);
```

Function: `tan` computes the tangent of *x*, expressed in radians.

tanh

Syntax:

```
#include <math.h>
double tanh(double x);
```

Function: `tanh` computes the hyperbolic tangent of *x*.

time

Syntax:

```
#include <time.h>
time_t time(time_t *timer);
```

Function: `time` returns the current calendar time, expressed as the number of seconds that have elapsed since Jan 1, 1970 12:00 A.M.

Notes: `time` must be implemented by the user. The current return value is (`time_t`)–1.

tmpfile

Syntax:

```
#include <stdio.h>
FILE *tmpfile(void);
```

Function: `tmpfile` creates a file that will be automatically be removed when the file is closed or the program exits.

Notes: `tmpfile` must be implemented by the user.

tmpnam

Syntax:

```
#include <stdio.h>
char *tmpnam(char *s);
```

Function: `tmpnam` returns a valid file name that will not conflict with any existing file names.

tolower

Syntax:

```
#include <ctype.h>
int tolower(int c);
```

Function: `tolower` converts an upper–case letter *c* to its lower–case equivalent, leaving all other characters unmodified. `tolower` returns the corresponding lower–case character or the unchanged character.

Notes: `tolower` is a function, not a macro. If *c* is known to be an upper–case character, then `__tolower` is faster.

toupper

Syntax:

```
#include <ctype.h>
int toupper(int c);
```

Function: `toupper` converts a lower–case character *c* to its upper–case equivalent, leaving all other characters unmodified. `toupper` returns the corresponding upper–case letter or the unchanged letter.

Notes: `toupper` is a function, not a macro. If *c* is known to be a lower–case character, then `__toupper` is faster.

ungetc

Syntax:

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Function: `ungetc` "puts back" the character *c* into the specified input *stream*.

Notes: The ANSI C standard only guarantees that one character can be "ungotten" without an intervening read.

va_arg

Syntax:

```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

Function: The `va_arg` macro expands to an expression that has the type *type* and value of the next varying argument in the call. The parameter *ap* must be the same as that returned by `va_start`. Each invocation of `va_arg` modifies *ap* so that the values of successive arguments are returned.

va_end

Syntax:

```
#include <stdarg.h>
void va_end(va_list ap);
```

Function: The `va_end` macro is used after all parameters on a variable length parameter list have been accessed with `va_arg`. It must be referenced before return from the function that contains the variable length argument list.

Notes: `va_end` generates no code; it is only used to guarantee portability to other compiler systems.

va_start

Syntax:

```
#include <stdarg.h>
void va_start(va_list ap, parameter);
```

Function: The macro `va_start` is used to initialize the reading of variable length arguments. It initializes *ap* for subsequent use by `va_arg` and `va_end`. *parameter* is the rightmost identifier in the variable parameter list in the function definition (the one just before the ellipsis).

vfprintf

Syntax:

```
#include <stdio.h>
int vfprintf(FILE *stream, const char
    *format, va_list arg);
```

Function: `vfprintf` is equivalent to `fprintf`, with the variable argument list replaced by *arg*. *arg* must be initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). `vfprintf` returns the number of characters transmitted or a negative number if an output error occurred.

Notes: `vfprintf` does not invoke the `va_end` macro.

vprintf

Syntax:

```
#include <stdio.h>
int vprintf(const char *format, va_list arg);
```

Function: `vprintf` is equivalent to `printf`, with the variable argument list replaced by *arg*. *arg* must be initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). `vprintf` returns the number of characters transmitted or a negative number if an output error occurred.

Notes: `__yvprintf` and `vprintf` are identical.

`vprintf` does not invoke the `va_end` macro.

vsprintf

Syntax:

```
#include <stdio.h>
int vsprintf(char *s, const char *format,
   va_list arg);
```

Function: `vsprintf` is equivalent to `sprintf`, with the variable argument list replaced by *arg*. *arg* must be initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). `vsprintf` returns the number of characters written to the array, not including the terminating null character.

Notes: `vsprintf` does not invoke the `va_end` macro.

wcstombs

Syntax:

```
#include <stdlib.h>
size_t wcstombs(char *s, const wchar_t
    * pwcs, size_t n);
```

Function: `wcstombs` copies *n* wide characters from *pwcs* to the multi–byte character string *s*. `wctombs` returns 0 if *pwcs* is null, and returns –1 if pwcs contains an invalid character. Otherwise, up to *n* characters are copied from *pwcs* to *s* and the number of characters copied is returned. The `wcstombs` routine assumes the Shift JIS convention for Japanese characters. See `mblen` for more details.

wctomb

Syntax:

```
#include <stdlib.h>
int wctomb(char *s, wchar_t wc);
```

Function: `wctomb` copies the wide character *wc* to *s*. It returns 0 if `s` is a null pointer and returns –1 if *wc* is an invalid character. Otherwise *wc* is written at *s* and 1 is returned if the character is a one–byte ASCII or kana code and 2 is returned if it is a two–byte Kanji code. `wctomb` assumes the shift JIS convention for Japanese characters. See `mblen` for more details.

CHAPTER 3

The Assembly Language Reference

The assembly language implemented by the InterTools 68000 Family assembler was designed by Motorola, Inc. It has the features commonly found in modern macro assembly languages. These features include absolute/relocatable code generation, complex relocatable expressions, macros, conditional assembly, and structured syntax.This chapter summarizes the basic structure of the assembly language, and gives an overview of the assembly language features.

3.1 Preface

The 68000 family assembler translates assembly language source programs into object modules. It is part of InterTools, an integrated set of cross–compilers, assemblers, source level debuggers and other utilities. The other parts of the InterTools System are described in the *InterTools User's Manual*, which gives information on how to invoke the assembler on your host system.

The InterTools 68000 assembler was designed to be compatible with the Motorola 68000 assembler. Most programs developed for the Motorola assembler should be readily portable and source–compatible with the InterTools 68000 assembler.

This document provides information necessary to use the assembler to develop assembly language programs for the Motorola 68000 family of microprocessors. It is not a comprehensive guide to the instruction set and architecture of the 68000 family of microprocessors.

Portions of this document are copyrighted by and used with the permission of Motorola, Inc.

3.2 Related Publications

The following Motorola Inc. publications provide a comprehensive treatment of microprocessor architecture and the instruction set, including Floating Point Coprocessor and Paged Memory Management Unit Coprocessor operation. They may be ordered from

the Literature Distribution Center, Motorola Inc., 616 West 24th Street, Tempe, AZ 85282 (602) 994–6561.

- *MC68040 32–Bit Microprocessor User's Manual*
  MC68040UM/AD REV 1
- *MC68881/MC68882 Floating Point Coprocessor User's Manual*
  MC68881/MC68882UM/AD
- *MC68851 Paged Memory Management Unit Coprocessor User's Manual*
  MC68851UM/AD
- *M68300 Family CPU32 Central Processor Unit Reference Manual*
  CPU32RM/AD REV 1
- *M68332 User's Manual*
  MC68332UM/AD
- *MC68302 Integrated Multiprotocol Processor User's Manual*
  MC68302UM/AD REV 2
- *MC68340 Integrated Processor with DMA User's Manual*
  MC68340UM/AD REV 1
- *MC68360 Quad Integrated Communications Controller User's Manual*
  MC68360UM/AD
- *MC68000 Family Programmer's Reference Manual*
  M68000PM/AD
- *MC68060 32–bit Microprocessor User's Manual*
  MC68060UM/AD

See also Motorola's Web Site for additional technical info:
www.mot.com/SPS

3.3 Using Assembly Language

High–level languages (e.g., C and Pascal) can decrease the development time needed for a program or system. All things being equal, a high–level language program is likely to be more reliable, easier to understand, and easier to maintain. However, high–level languages do not permit the programmer to directly access all the microprocessor's features, such as registers, processor flags, and special instructions.

As a result, an assembly language program, while sometimes taking longer to code and debug, can run much faster and occupy less memory than the equivalent program written in a high–level language. Since program development time is expensive, the trade–off between development time and program performance should be analyzed for each application. The optimal solution is usually found in writing most routines in a high–level language and in writing the time–critical, space–critical, and special routines (e.g., I/O routines) in assembly language.

3.4 Elements of Assembly Language

The lines in an assembly language source file can be classified in four general categories: instruction statements, data allocation statements, assembler directives, and assembler controls.

An **instruction statement** uses an easily remembered name, a **mnemonic**, and possibly one or two operands+ to specify a machine instruction to be generated.

A **data allocation statement** reserves, and optionally initializes, memory space for program data.

An **assembler directive** is a statement that gives special instructions to the assembler. Although directives may produce something in the object file, they are unlike the instruction and data allocation statements in that they do not specify the actual contents of memory.

An **assembler control** is also a statement that gives special instructions to the assembler. Assembler controls are used to control the assembly process rather than to define the meaning of the program being assembled.

Here are some examples of the different kinds of statements:

Statement Type	Examples
Instruction	`MOVE D2,D4` `JSR SORT_PROCEDURE`
Data Allocation	`DS.W 0` `DC.B 'H'`
Assembler Directive	`SECTION MYSEC` `COUNT EQU 5`
Assembler Control	`NOPAGE` `INCLUDE source.inc`

In addition, the language is composed of the following symbolic elements:

- Symbolic names or labels, which are instructions, directives, register mnemonics, user–defined memory labels, and macros.
- Numbers, which may be represented in binary, octal or decimal.
- Arithmetic and logical operators, which are used in complex expressions.
- Special purpose characters, which are used for macro functions, source line fields, and numeric bases.

3.5 Notation

A small amount of specialized notation is used in this document to specify the general format for instructions and directives. It is based on fairly standard additions to the Backus–Naur Form (BNF) formalism. The four "metasymbols" described below are used throughout this manual to indicate that the user must replace the metasymbols and the characters they enclose with some legal text. The actual text that can be substituted will be different in each case, and depends on what type of assembly language statement is being described.

When symbols or metasymbols represent the actual commands or text to be supplied, they will be in **boldface**. When symbols or metasymbols must be replaced by commands or text, they will be in *italics*. In some cases, the characters used as metasymbols are required characters in a statement rather than metasymbols. In those cases they will appear in boldface rather than italics. The four metasymbols and their meanings are:

< >	Angular brackets enclosing a name indicate that one element of the general category specified by the name is to be selected.

|	The pipe (vertical bar) indicates that a choice is to be made. One of the symbols separated by the pipe character(s) should be selected.
[ ]	Square brackets indicate that the enclosed sequence is optional. The enclosed sequence may occur one time or not at all.
[ ]...	Square brackets followed by periods enclose a symbol that is optional but repetitive. The symbol may appear zero or more times.

For example,

MOVE*[.<size>]* *<source>,<destination>*

where:

<size> = *B* | *W* | *L*

BFEXTS *<ea>*{*<offset>*:*<width>*},**D***n*

where:

<offset> = #*<expr>* | **D***n*
<width> = #*<expr>* | **D***n*

In the first example, the *[.<size>]* notation implies the user may supply MOVE.L, MOVE.W, MOVE.B, or MOVE as legal mnemonics.

In the second example, the user must supply either an immediate value or a data register for the "offset" and "width" symbols. The curly braces must appear in the instruction. Thus, a legal instruction would be:

BFEXTS LAB{0:8},D1.

CHAPTER 4

Source Program Coding

A source program is a sequence of source statements arranged in a logical way to perform a predetermined task. The assembler interprets and processes each source line, generating object code or performing a specific assembly time process. This chapter discusses all facets of source program coding including source line format, symbols, constants, registers, operators, expressions, addressing modes, instruction mnemonics, and other instruction types.

4.1 Introduction

A source program is a sequence of source statements arranged in a logical way to perform a predetermined task. The assembler interprets and processes each source line, generating object code or performing a specific assembly time process. Each source statement occupies a line of printable text, where each line may be one of the following:

- Comment
- Executable instruction
- Assembler directive
- Macro invocation

Note: The InterTools Motorola 68000 series cross assemblers are case insensitive to source input except as noted under the INCLUDE directive or for ASCII strings.

4.2 Comments

Comments are strings of characters that are inserted only to identify or clarify the program. Comments are included in the assembly listing but are otherwise ignored by the assembler.

A comment may be inserted in one of two ways:

- As a line, starting in column one, where an asterisk (*) is the first character in the line. The entire line is a comment, and an instruction or directive in this line will not be recognized.
- Following the operation and operand fields of an assembler instruction or directive, where it is preceded by at least one blank or tab character.

Example

```
* THIS ENTIRE LINE IS A COMMENT.
      BRA LAB2    THIS COMMENT FOLLOWS AN INSTRUCTION
```

4.3 Source Line Format

Each source statement has an overall format that is some combination of the following fields:

- Label
- Operation
- Operand
- Comment

The statement lines in the source file must not be numbered. However, in the listing file the assembler prefixes each line with a sequential number, up to four decimal digits.

The format of each line of source code is described in the following paragraphs.

4.3.1 Label Field

The label field is the first field in the source line. A label which begins in the first column of the line may be terminated by either a blank, a tab, or a colon. A label may be preceded by one or more blanks or tabs, provided it is then terminated by a colon. In either case, the colon is not considered to be part of the label.

Labels are allowed on all instructions and on all assembler directives which define data structures. For such operations, the label is defined with a value equal to the location counter for the beginning of the instruction or directive, including a designation for the program section (if there is one) in which the definition appears.

Labels are required on the assembler directives which define symbol values (SET, EQU, REG). For these directives, the label is defined to be the value of the expression in the operand field. This value consists of a constant and, if the expression is relocatable, a section designation.

Labels are required on macro definitions and serve as the mnemonic by which that macro is subsequently invoked. No memory address is associated with such labels. A label is also required on the IDNT directive. This label is passed on to the relocatable object module *but it has no associated internal value*. The IDNT statement, therefore, cannot be used to define program entry points.

Labels which are the only field in the source line are defined equal to the current location counter value plus, if the section is relocatable, the program section.

4.3.2 Operation Field

The operation field follows the label field and is separated from it by at least one blank or tab. If the label field is not being used, the operation field must be at least one blank or tab from the left margin. The operation field specifies the action to be performed by the statement. Entries in this field fall under one of the following categories:

- *Instruction mnemonics*: The M68000–family processor instruction set.
- *Directive mnemonics.*: These control the assembly process.
- *Macro calls.*: These are invocations of previously defined macros.

The size of the data field affected by an instruction is determined by the data size code. Some instructions and directives can operate on more than one data size. For these operations, the data size code must be specified or a default size is assumed. The size code need not be specified if only one data size is permitted by the operation. The data size code is specified by appending a period (.) to the operation field, followed by B, W, L, S, D, X, or P, where the appended character is interpreted as described below:

B Byte (8–bit data)

W Word (16–bit data)

L Longword (32–bit data)

S Byte (8–bit signed offset for certain branch instructions)

Single precision binary real (IEEE Standard, 32–bit: 1–bit sign, 8–bit exponent, 23–bit mantissa) (68881/68882/68040/68060 only)

D Double word (64–bit data) (68030, 68040, 68060, and 68851 only)

Double precision binary real (IEEE Standard, 64–bit: 1–bit sign, 11–bit exponent, 52–bit mantissa) (68881/68882/68040/68060 only)

X Extended precision binary real (96–bit: 15–bit exponent, 1–bit sign, 64–bit mantissa), (16 bits are reserved) (68881/68882/68040/68060 only)

P Packed Binary Coded Decimal (BCD) real string (96–bit: 3–decimal digit exponent and 17–decimal digit mantissa) (68881/68882/68040/68060 only)

The data size code is not permitted, however, when the instruction or directive does not have a data size attribute.

Examples (legal)

```
    LEA 2(A0),A1    Longword size is assumed.
*                   (.B, .W not allowed) This
*                   instruction loads the
*                   effective address pointed
*                   to by A0+2 into A1.
*
    ADD.B ADDR,D0   This adds the byte whose
*                   address is ADDR to
*                   the low order byte in DO.
*
    ADD D1,D2       This adds the low order
*                   word of D1 to the low
*                   order word of D2. (W is
*                   the default size code.)
*
    ADD.L A3,D3     This adds the 32-bit
*                   contents of A3 to D3.
*
```

Example (illegal)

```
    SUBA.B #5,A1    Illegal size specification
*                   (.B not allowed on SUBA)
*                   This instruction attempts
*                   to subtract the value 5
*                   from the low order byte
*                   of A1; but byte operations
*                   on address registers are
*                   not allowed.
*
```

4.3.3 Operand Field

If present, the operand field follows the operation field and is separated from the operation field by at least one blank or tab. When two or more operand subfields appear within a statement, they must be separated by a comma but may not contain embedded blanks or tabs; e.g., "ADD D1, D2" is illegal.

For most two operand instructions, the general format "*opcode source,destination*" applies. For example, in an instruction like "ADD D1,D2", the contents of D1 are added to the contents of D2 and the result is saved in register D2. In the instruction "MOVE D1,D2", the first operand (D1) is the sending operand and the second operand (D2) is the receiving operand.

4.3.4 Comment Field

The last field of a source statement is an optional comment field. This field is ignored by the assembler, but is included in the listing. The comment field is separated from the operand field (or the operation field, if there is no operand) by at least one blank or tab, or by a semicolon (;), and may consist of any ASCII characters.

4.4 Symbols

Symbols can correspond to either a specific numerical value by using an EQU or SET directive, or the address of a memory location. The memory location can represent the destination of a branch instruction or the start of a data area. This use of symbolic references to memory allows statements to be written without specifying actual memory locations. An entry in the label field is required for all statements that are the destination of jump and branch instructions and in statements using the EQU or SET directives.

4.4.1 Symbol Syntax

Symbols recognized by the assembler consist of one or more valid characters (refer to the *Character Set* appendix), all of which are significant. The first character must be an upper case or lower case letter, (A–Z or a–z), a period (.) or an underscore (_). Each remaining character may be an upper case or lower case letter (A–Z, a–z), a period (.), an underscore (_), a digit (0–9), a dollar sign ($), or a question mark (?).

4.4.2 Symbol Definition Classes

Symbols may be differentiated by usage into two classes. Class 1 symbols are used in the operation field of an instruction. Class 2 symbols occur in the label and operand fields of the instruction. Assembler directives (including those for structured assembly), instruction mnemonics, and macro names comprise Class 1 symbols; Class 2 symbols consist of user defined labels and register mnemonics.

A Class 1 symbol may be redefined and used independently as a Class 2 symbol, and vice versa. As long as each symbol is used correctly, no conflict will result from the existence of two symbols of different classes with the same name. For example, the following is a legal instruction sequence:

```
ADD   D1,ADD
.
.
.
ADD   DS    2
```

By its usage as a Class 1 symbol, the first ADD is recognized as an instruction mnemonic; the second ADD is recognized as a Class 2 symbol identifying a reserved storage area. The assembler differentiates a Class 1 symbol from a Class 2 symbol with the same name, thereby allowing two symbol table entries with the same name but different class.

Macro labels are a special case because the same symbol will appear as the label (Class 2) in the MACRO definition and, subsequently, as an operation code mnemonic (Class 1) in invocation of that same macro. Macro labels are defined to be Class 1 symbols; their presence in the label field of a MACRO directive is ignored as a Class 2 symbol. Therefore, macro names may be redefined as Class 2 symbols without conflict.

Except for the SET directive, which allows multiple redefinition of a Class 2 symbol, a symbol may not be redefined within the same class. For example, SUB (reserved Class 1 symbol) may not be redefined as a macro label (also Class 1), nor may A5 (reserved Class 2 symbol) be redefined as a statement or storage location label (also Class 2).

4.4.3 User–Defined Labels

Labels are symbols that are defined by the user to identify memory locations in program or data areas of the assembly module.

Labels may have an absolute or relocatable value, depending upon the section in which the labeled memory location is found. If the memory location is within a relocatable section, the label has a relocatable value, i.e., it depends on where the section is placed. If the memory location is in an absolute section, the label has an absolute value.

Labels may be defined in the label field of an executable instruction or a data definition directive source line. It is also possible to define a label with the SET or EQU directive to an arbitrary value.

4.4.4 Location Counter Symbol "*"

The special symbol "*" may be used to refer to the current location counter value.

4.5 Constants

4.5.1 Integer Constants

Numeric constants recognized by the assembler may be expressed in decimal, hexadecimal, octal, and binary form. They must have integral values and must be expressible in 32 bits.

Decimal Constants

Decimal is the default base used for evaluating numeric values and consists of a string of numeric digits.

Example

```
        12345       Valid
        12.3        Invalid: can consist only of
*                   digits
```

Hexadecimal Constants

A hexadecimal constant consists of characters from the set of decimal digits (0–9) and the alphabetic characters (A–F, a–f) and is preceded by a dollar sign ($) or followed by an **H**.

If the suffix form is used, the first character must be a digit to distinguish the hexadecimal constant from a symbol name.

Example

```
        $12         Valid
        01CFH       Valid
        $01CF       Valid
        ABCDH       Invalid: no preceding $
*                   This would be interpreted as
*                   symbol ABCDH
```

Octal Constants

An octal constant consists of characters from the set of digits (0–7), preceded by a commercial at sign (@) or followed by a **Q**.

Example

```
        @17634      Valid
        275Q        Valid
        @27832      Invalid: character 8 not
*                   allowed
```

Binary Constants

A binary constant consists of 1's and 0's, preceded by a percent sign (%) or suffixed by a **B**.

Example

```
        %10100      Valid
        10111B      Valid
        %21001      Invalid: character 2 not
*                   allowed
```

4.5.2 Character Constants

One or more printable ASCII characters enclosed by apostrophes (') constitute a character constant. Character constants longer than four characters may only be stored in memory, e.g., using the DC (define constant) directive. Shorter character constants may also be used as immediate operands, in which case they are treated as an integer according to the rules described below.

Character constants are left justified and zero filled, if necessary, whether stored or used as immediate operands. Constants with four or three characters are aligned to a longword. Constants with two characters are aligned to a word. Single character constants are word aligned if the operand size is larger than a byte, and byte aligned if the operation size is a byte.

In order to specify an apostrophe within a character constant, two successive apostrophes must appear where the single apostrophe is intended to appear.

Examples

```
B_DIG_1   EQU.B     '1'          Equates B_DIG_1 with
*                                   hex 31
W_DIG_1   EQU       '1'          Equates W_DIG_1 with
*                                   hex 3100
          DC.L      '79'         Stores hex 37390000
*                                   in memory
          MOVE.L    #'1',D0      Moves hex
*                                   00003100 into D0
          MOVE.B    #'1',D0      Moves hex 31 into
*                                   low byte of D0
          MOVE.L    #'123',D0    Moves hex 31323300
*                                   into D0
```

4.5.3 Floating Point Constants (68881/68882/68040/68060 only)

IEEE standard floating point numbers can be specified by an optionally signed fraction string of up to 17 decimal digits (0–9) containing a required decimal point, the constant "E" an optional sign, and an exponent up to 3 decimal digits. The exponent section "E<sign>yyy" is optional; underscores can occur for readability.

Floating Point Constant Notation

[<*sign*>]*x.xxxxxxxxxxxxxxx*[**E**[<*sign*>]*yyy*]
(*maximum size*)

where:

<*sign*> is + | –
x and *y* are decimal digits

Example

```
DC.X     1234.56E-33
DC.X     -12345.67
```

Floating point numbers can also be specified explicitly as a series of hexadecimal digits preceded by a colon (:). This floating point hex format can be used to exactly represent the mantissa, exponent, and sign bit for a given floating point number.

Floating Point Hex Constant Notation

:*hhhhh*[*h*]...

where:
h is a hex digit

Up to 8 digits are allowed for .S precision, up to 16 for .D, and up to 24 for .X or .P.

Example

```
        DC.S        :124F67B    STORES HEX 124F67B0
*                               IN MEMORY
```

4.6 Registers

The 68000 has sixteen 32–bit registers (D0–D7, A0–A7) in addition to a 24–bit program counter and 16–bit status register.

Registers D0–D7 are used as data registers for byte, word, and longword operations. Registers A0–A7 are used as software stack pointers and base address registers; they may also be used for word and longword data operations. All 16 registers may be used as index registers.

Register A7 is used as the system stack pointer. The 68000 processor actually provides two hardware stack pointers, depending upon whether the instruction is executing in the supervisor or user state. Stack pointers and the supervisor/user states are explained under *Stacks and Queues* and *Privilege States* in the *68000 16/32–Bit Microprocessor Programmer's Reference Manual.*

The 68000–family processors fall into three groups:

- The 68000, 68302, and 68010 which have the basic instruction set.
- The 68020, 68030, 68040, 68060, 68EC030, and 68EC040 which have the full instruction set. We will refer to this group as **68020–plus** processors.
- The 68332 and 68340 which have more than the basic instruction set, but less that the full instruction set. We will refer to this group as **CPU32** processors.

There are two floating–point coprocessors, the 68881 and 68882, most of whose functions are provided on the 68040 and 68060. There is also a 68851 memory management coprocessor, some of whose functions are provided on the 68030, 68EC030, 68040, 68060 and 68EC040.

(For non–68000 and non–68302 only) Processors other than the 68000 and 68302 have these additional registers:

— one 32–bit register (VBR)
— two 3–bit registers (SFCR and DFCR)

The contents of the VBR are added to the previously calculated vector offset, during exception processing, to produce the actual vector location. The 3–bit registers allow supervisor access to other address spaces via MOVES, supplying function codes in the SFCR for the read cycle(s) from the effective address location, or supplying function codes in the DFCR for the write cycle(s) to the effective address location, respectively.

(For 68020–plus only) These processors have these additional registers:

— two 32–bit stack pointer registers (MSP and ISP)
— two 32–bit cache registers (CACR and CAAR)

MSP (master stack pointer) is active whenever both the "S" and "M" bits of the status register are set (supervisor state). ISP (interrupt stack pointer) is active whenever the "S" bit is set but not the "M" bit (interrupt state). USP (user stack pointer) is active whenever the "S" bit is 0 (user state). The cache registers support the on board instruction caches of the 68020–plus and can be accessed only in the supervisor state.

(For 68020–plus and CPU32 only) The assembler also provides 17 "pseudo register" names for the CPU32 and 68020–plus. These are the

suppressed address registers used in various 68020–plus and CPU32 addressing modes. Each register represents a content value of zero:

ZA0–ZA7 Suppressed address registers
ZD0–ZD7 Suppressed data registers
ZPC Suppressed program counter

(For 68030 only) The 68030 has the following additional registers:
— two 32–bit transparent translation registers (TT0 and TT1)
— one 16–bit MMU status register (MMUSR)
— one 64–bit CPU root pointer (CRP)
— one 64–bit supervisor root pointer (SRP)
— one 32–bit translation control register (TC)
The MMUSR, SRP, and TC registers also exist on the 68040.
The CRP, SRP, and TC registers also exist on the 68551.

(For 68040 only) The 68040 has the following additional registers:
— two 32–bit data transparent translation registers (DTT0 and DTT1)
— two 32–bit instruction transparent translation registers (ITT0 and ITT1)
— one 16–bit MMU status register (MMUSR)
— one 32–bit user root pointer (URP)
— one 32–bit supervisor root pointer (SRP)
— one 16–bit translation control register (TC)
The MMUSR, SRP, and TC registers also exist on the 68030.
The SRP and TC registers also exist on the 68851.

(For 68881/68882/68040 only) The 68881/68882/68040 floating point operations use the following registers:

— eight 80–bit registers (FP0–FP7)

— three 32–bit registers (STATUS, CONTROL, and IADDR)

The 80–bit registers are the floating point data registers, FP0–FP7, that serve as destinations for most floating point operations. The 32–bit registers are the system registers STATUS, CONTROL, and IADDR. These registers may also be referenced as FPSR, FPCR, and FPIAR. These control rounding and the treatment of floating point exceptions.

(For 68851 only) The 68851 paged memory management unit coprocessor provides these additional registers:

— one 64–bit DMA translation tables root pointer (DRP)

— one 16–bit cache status register (PCSR)

— one 16–bit status register (PSR)

— three 8–bit protection control registers (CAL, VAL, and SCC)

— one 16–bit access control register (AC)

— eight 16–bit breakpoint acknowledge data registers (BAD0–BAD7

— eight 16–bit breakpoint acknowledge control registers (BAC0–BAC7)

— one 64–bit CPU root pointer (CRP)

— one 64–bit supervisor root pointer (SRP)

— one 32–bit translation control register (TC)

The CRP, SRP, and TC registers also exist on the 68030.
The SRP and TC registers also exist on the 68040.

(For 68EC030 only) The 68EC030 has the following additional registers:

— two 32–bit access control registers (AC0 and AC1)

— one 16–bit ACU status register (ACUSR)

Note that the ACUSR register also exists on 68EC040.

(For 68EC040 only) The 68EC040 has the following additional registers:

— two 32–bit data access control registers (DACR0 and DACR1)
— two 32–bit instruction access control registers (IACR0 and IACR1)
— one 16–bit ACU status register (ACUSR)

Note that the ACUSR register also exists on 68EC030.

(For 68060 only) The 68060 has the following additional registers:

— two 32–bit data transparent translation registers (DTT0 and DTT1)
— two 32–bit instruction transparent translation registers (ITT0 and ITT1)
— one 32–bit user root pointer (URP)
— one 32–bit supervisor root pointer (SRP)
— one 16–bit translation control register (TC)
— one 32–bit processor configuration register (PCR)
— one 32–bit bus control register (BUSCR)

The SRP and TC registers also exist on the 68030.
The SRP and TC registers also exist on the 68851.

The following register mnemonics are recognized by the assembler:

D0–D7	Data registers.
ZD0–ZD7	Suppressed data registers (68020–plus and CPU32 only).
A0–A7	Address registers.
ZA0–ZA7	Suppressed address registers (68020–plus and CPU32 only).
A7, SP	Either mnemonic represents the system stack pointer of the active system state.
PC	Program counter. Used only in forcing program counter relative addressing.

ZPC	Suppressed program counter (68020–plus and CPU32 only).
CCR	Condition code register (low 8 bits of SR).
SR	Status register. All 16 bits may be modified in the supervisor state. Only low 8 bits (CCR) may be modified in user state.
SSP	Supervisor stack pointer (basic 68000, 68060, and CPU32 only).
USP	User stack pointer (for user state on the (68020–plus, CPU32, 68060 and 68010 only).
ISP	Interrupt stack pointer (for interrupt state on 68020–plus only).
MSP	Master stack pointer (for supervisor state on 68020–plus only).
VBR	Vector base register (68010, 68020–plus, and CPU32 only). Supports multiple vector table areas during exception processing. Accessed by the MOVEC instruction.
SFC or SFCR	Alternate function code source register (68010, 68020–plus, and CPU32 only). Accessed by the MOVEC instruction).
DFC or DFCR	Alternate function code destination register (68010, 68020–plus, and CPU32 only). Accessed by the MOVEC instruction.
CACR	Cache control register (68020–plus only). This holds the address of the cache control functions requiring an address.
CAAR	Cache address register (68020, 68030, and 68EC030 only). This holds the address of the cache control functions requiring an address.

CRP CPU root pointer (68030 and 68851 only). This holds a pointer to the root of the translation tree for the task that is currently executing.

SRP Supervisor root pointer (68030, 68040, 68060, and 68851 only). This holds a pointer to the root of the translation tree for the supervisor's address space.

TC Translation control register (68030, 68040, 68060, and 68851 only). This register has several fields controlling address translation.

TT0, TT1 Transparent translation registers (68030 only). These can be used to identify separate memory blocks as being directly accessible without address translation.

MMUSR MMU status register (68030 and 68040 only). This holds memory management status information.

DTT0, DTT1, ITT0, ITT1
The data and instruction transparent translation registers (68040 and 68060 only). These can be used to identify separate memory blocks as being directly accessible without address translation.

URP User root pointer (68040 and 68060 only). This holds a pointer to the root of the translation tree for the user's address space.

AC0, AC1 Access control registers (68EC030 only). These are used to define two memory spaces with caching restrictions.

ACUSR ACU status register (68EC030 and 68EC040 only). This register is used to show the access control information resulting from a test of the access control registers.

DACR0, DACR1, IACR0, IACR1
Access control registers (68EC040 only). These can each specify separate blocks of memory as accessible with restrictions.

DRP DMA translation table root pointer (68851 only).

PCSR PMMU cache status register (68851 only). This holds information about the internal translation cache.

PSR PMMU status register (68851 only). This holds information about the status and access rights for a specific logical address.

CAL Current access level register (68851 only).

VAL Validate access level register (68851 only).

SCC Stack change control register (68851 only).

AC Access control register (68851 only).

BAD0–BAD7 Breakpoint acknowledge data registers (68851 only). These contain the opcodes that are given to the CPU during a breakpoint acknowledge cycle.

BAC0–BAC7 Breakpoint acknowledge control registers (68851 only). These hold the enable and count functions for the instruction breakpoint acknowledge mechanism.

FP0–FP7 Floating point data registers (68881, 68882, 68060 and 68040 only).

CONTROL or FPCR

Floating point control register (68881, 68882, 68060 and 68040 only). Contains four bytes. The third is the exception enable byte to enable/ disable traps for each class of floating point exception. The fourth byte is a mode byte to set the user selectable modes. The remaining bytes are reserved.

8STATUS or FPSR Floating point status register (68881, 68882, 68060 and 68040 only). Contains four bytes. The first byte is a floating point condition code byte, containing five condition codes that are set by all move and arithmetic floating point instructions except FMOVEM. The third byte is a floating point exception byte. The fourth byte is a floating point accrued exception byte containing the logical inclusive OR of all floating point exceptions that have occurred since this byte was last cleared by the user. The remaining bytes are reserved.

IADDR or FPIAR Floating point instruction address register (68881, 68882, 68060 and 68040 only). Contains the logical address in the main processor memory of the offending instruction that generated a floating point exception trap.

BOSCR Bus control register (68060 only). Contains bits which control the internal pipelines.

PCR Processor configuration register (68060 only). Contains bit to control bus interface controller.

	basic 68000			68020–plus						CPU32		MMU
regs	68000	68302	68010	68020	68030	68040	EC030	Ec040	68060	68332–68340		68851
USER PROGRAM MODEL												
D0–D7	X	X	X	X	X	X	X	X	X	X	X	–
ZD0–ZD7	–	–	–	X	X	X	X	X	X	X	X	–
A0–A6	X	X	X	X	X	X	X	X	X	X	X	–
ZA0–ZA7	–	–	–	X	X	X	X	X	X	X	X	–
A7 (USP)	X	X	X	X	X	X	X	X	X	X	X	–
PC	X	X	X	X	X	X	X	X	X	X	X	–
CCR	X	X	X	X	X	X	X	X	X	X	X	–

	basic 68000			68020–plus						CPU32		MMU
regs	68000	68302	68010	68020	68030	68040	EC030	EC040	68060	68332	68340	68851
SUPERVISOR PROGRAMMING MODEL												
SR	X	X	X	X	X	X	X	X	X	X	X	–
A7 (SSP)	X	X	X	–	–	–	–	–	X	X	X	–
A7 (ISP)	–	–	–	X	X	X	X	X	–	–	–	–
A7 (MSP)	–	–	–	X	X	X	X	X	–	–	–	–
VBR	–	–	X	X	X	X	X	X	X	X	X	–
SFCR	–	–	X	X	X	X	X	X	X	X	X	–
DFCR	–	–	X	X	X	X	X	X	X	X	X	–
CACR	–	–	–	X	X	X	X	X	X	–	–	–
CAAR	–	–	–	X	X	–	X	–	–	–	–	–
CRP	–	–	–	–	X	–	–	–	–	–	–	X
SRP	–	–	–	–	X	X	–	–	X	–	–	X
TC	–	–	–	–	X	X	–	–	X	–	–	X
TT0,TT1	–	–	–	–	X	–	–	–	–	–	–	–
MMUSR	–	–	–	–	X	X	–	–	–	–	–	–
DTT0, DTT1	–	–	–	–	–	X	–	–	X	–	–	–
ITT0, ITT1	–	–	–	–	–	X	–	–	X	–	–	–
URP	–	–	–	–	–	X	–	–	X	–	–	–
AC0, AC1	–	–	–	–	–	–	X	–	–	–	–	–
ACUSR	–	–	–	–	–	–	X	X	–	–	–	–
DACR0, DACR1	–	–	–	–	–	–	–	X	–	–	–	–
IACR0, IACR1	–	–	–	–	–	–	–	X	–	–	–	–
DRP	–	–	–	–	–	–	–	–	–	–	–	X
PCSR	–	–	–	–	–	–	–	–	–	–	–	X
PSR	–	–	–	–	–	–	–	–	–	–	–	X
CAL, VAL, SCC	–	–	–	–	–	–	–	–	–	–	–	X
AC	–	–	–	–	–	–	–	–	–	–	–	X
BAD0– BAD7	–	–	–	–	–	–	–	–	–	–	–	X
BAC0– BAC7	–	–	–	–	–	–	–	–	X	–	–	–
PCR	–	–	–	–	–	–	–	–	X	–	–	–
BUSCR												

	68020–plus						floating point coprocessors	
regs	**68020**	**68030**	**68040**	**EC030**	**EC040**	**68060**	**68881**	**68882**
FP0–FP7	–	–	X	–	–	X	X	X
FPCR	–	–	X	–	–	X	X	X
FPSR	–	–	X	–	–	X	X	X
FPIAR	–	–	X	–	–	X	X	X

4.7 Operators

Operators recognized by the assembler include, in order of operator precedence (from highest to lowest):

1. Parenthetical Expression (innermost first)
2. Unary Minus, Plus, `SEGSIZE`, `SEGBASE`
3. Shift Right, Shift Left
4. And, Or
5. Multiplication, Division, Remainder
6. Addition, Subtraction

Expressions involving operators with the same precedence are evaluated from left to right. All expressions are evaluated using 32–bit values and all intermediate results are stored internally as 32–bit integers.

All examples are shown using constants. Absolute labels, however, may be used in any expression where a constant is shown below. Absolute labels include labels of data or code located in absolute sections, or labels on EQU or SET directives which have an expression that evaluates to a constant value. In addition, labels on DS directives in the table associated with an OFFSET directive are absolute. Relocatable labels and `SEGSIZE` or `SEGBASE` values may only be used in expressions involving addition and subtraction.

Arithmetic Operators:		
addition	(+)	
subtraction	(–)	
multiplication	(*)	
division	(/)	Produces a truncated integer result.
remainder	(%)	
unary minus	(–)	
segment size	`SEGSIZE(`*seg*`)`	
segment base	`SEGBASE(`*seg*`)`	
Shift Operators (binary):		
shift right	(>>)	The left operand is shifted to the right (and zero filled) by the number of bits specified by the right operand.
shift left	(<<)	Analogous to >.
Logical Operators (binary):		
and	(&)	
or	(!)	

Note: The percent (remainder) operator is only valid if the "select TASKING extensions" option is in effect. See the *68000 Family Assembler* chapter in the *InterTools User's Manual* for more information about this option. The same is true of the `SEGSIZE` and `SEGBASE` operators.

SEGSIZE — Get Segment Size

Syntax:

```
SEGSIZE(<name>)
```

The SEGSIZE operator returns the size in bytes of the named section. It is necessary to XREF the section name if it is not defined in the current module.

SEGBASE — Get Base Address of Segment

Syntax:

```
SEGBASE(<name>)
```

The SEGBASE operator returns the base address of the named section. It is necessary to XREF the section name if it is not defined in the current module.

Example

```
        XREF udata

        SECTION foo,,"code"

        MOVEA.L #SEGBASE(udata),A0      load base address of udata
        MOVE.L  #SEGSIZE(udata)-1,D0    load size of udata - 1 (for dbf)
loop    CLR.B   (A0)+
        DBF     D0,loop                 zero out udata segment
```

4.8 Expressions

Expressions are composed of one or more symbols and/or constants that may be combined with unary or binary operations. Expressions are evaluated according to precedence rules left to right.

Subexpressions which involve relocatable symbols may use only the "+" and "–" operators. It is possible for a subexpression involving the difference between two relocatable symbols to evaluate to an absolute value. For example, let R1 represent a memory location at OFFSET1 bytes beyond the start of section S1, and let R2 represent a memory location at OFFSET2 bytes beyond the start of section S2. That is:

```
R1 = OFFSET1 + <start of S1>
R2 = OFFSET2 + <start of S2>
```

The difference between R1 and R2 may then be:

```
R1-R2 = OFFSET1-OFFSET2
+ <start of S1> - <start of S2>
```

If sections S1 and S2 are the same, then:

```
R1-R2 = OFFSET1-OFFSET2
```

which is an absolute (non–relocatable) value. Of course, if sections S1 and S2 are separate and distinct, the expression remains a complex relocatable expression.

When an expression has been fully evaluated by the assembler, it may be categorized as one of three types of expressions:

- *Absolute Expression*:
  The expression has reduced to an absolute value that is independent of the start address of any relocatable section.
- *Simple Relocatable Expression*:
  The expression has reduced to an absolute offset from the start of a single relocatable section.
- *Complex Relocatable Expression*:
  The expression has reduced to a constant, absolute offset in conjunction with either of the following relocatable terms:
  1. A single, negated start address of a relocatable section.
  2. An expression of two or more relocatable symbols, or containing a `SEGSIZE` or `SEGBASE` value.

Note: Only absolute expressions with no forward references are legal in ORG, OFFSET, DCB, and DS directives. SET and EQU can take any expression.

By themselves, all user–defined labels on memory locations are either absolute or simple relocatable expressions. This includes XREF labels, which are assumed to be absolute symbols unless their program section is specified. Complex relocatable expressions may arise only from the addition or subtraction of two relocatable expressions.

The following are examples of each type of expression:

```
          ORG       $1000          absolute section
ARRAY     DS        $20            ARRAY is absolute
ENDARRY   EQU       *-2            ENDARRY
*                                      is absolute
          SECTION   2              relocatable
*                                      table section
L1        CLR.L     D2             L1 is simple
*                                      relocatable
L2        MOVE      D3,(A0)        L2 is simple
*                                      relocatable
          MOVE.L    ARRAY+10,D7    absolute source
*                                      operand
          MOVE.L    L1+10,D7       simple relocatable
*                                      source operand
          MOVE.L    L2-L1,D7       absolute source
*                                      operand
          MOVE.L    L1+L2,D7       complex relocatable
*                                      source operand
```

4.9 Addressing Modes

Effective address modes, combined with operation codes, define the particular function to be performed by a given instruction. Effective addresses and data organization are described in detail in the *Data Organization and Addressing Capabilities* section of the *Microprocessor Manual* for the appropriate processor.

The 68000 and 68010 require that references to words or longwords be at an even byte boundary. The 68020–plus and CPU32 impose no such restriction, but will run faster if word and longword references are at a multiple of the operand size. Instructions must always begin on an even byte boundary.

Individual bits within a byte (operand for memory destinations) or longword (operands for data register destinations) may be addressed with the bit manipulation instructions. Bits for a byte are numbered 7 to 0, with 7 being the most significant bit position and 0 the least significant. Bits for a word are numbered 15 to 0, with 15 being the most significant bit and 0 the least significant. Bits for a longword are numbered from 31 to 0, with 31 being the most significant bit position and 0 the least significant bit position.

The code generated in the listing file for some addresses may be the same as the code generated for different expressions whenever externally referenced symbols are involved. The linked object file contains the correctly resolved addresses.

Definitions of the symbols used in Tables 2–1 and 2–2 and throughout the remainder of this manual are given below:

An	Address register number *n* (0–7).
ZAn	(CPU32/68020–plus only) Suppressed address register number *n* (0–7) whose value is taken to be zero. Can be used in place of An if suppression is desired.
Dn	Data register number *n* (0–7).

Rn	Address register number *n* or Data register number *n* (0–7).
Ri	Index register number *i*; may be any address (An) or data (Dn) register with optional .W or .L size designation (16 vs. 32 bits). On the 68020–plus and CPU32, a scaling factor *scl may also exist.
ZRi	(68020–plus and CPU32 only) Suppressed index register number *i* (0–7) whose value is taken to be zero. Can be used in place of Ri if suppression is desired.
scl	(68020–plus and CPU32 only) Scaling factor of 1, 2, 4, 8 optionally used in indexing modes. The default is 1.
PC	Program counter.
ZPC	(68020–plus and CPU32 only) Suppressed program counter whose value is taken to be zero. Can be used in place of PC if suppression is desired.
B,W,L	Byte, word, longword data sizes.
d(An)	Address register indirect with displacement (**d**).
d(An,Ri)	Address register indirect with index (**Ri**) plus displacement (**d**).
d(PC)	Program counter with displacement (**d**).
d(PC,Ri)	Program counter with index (**Ri**) plus displacement (**d**).
<absolute>	Absolute expression.
<simple>	Simple relocatable expression (such as a label or a symbol).

<complex>	Complex relocatable expression (involving arithmetic on simple relocatables).
bd	(68020–plus and CPU32 only) Base displacement that is added before indirection occurs.
od	(68020–plus and CPU32 only) Outer displacement that is added after indirection occurs. Displacement size may be either word or longword.
<ea>	Effective address expression.
<iea>	(68020–plus and CPU32 only) Indirect effective address expression.

Null Displacements (68020–plus and CPU32 only)

Null displacements imply that no extension word is present in the instruction for displacement.

Omitted Registers and Displacements (68020–plus and CPU32 only)

Omitted registers take on suppressed register values (taken to be zero). Omitted displacements take on null values (taken to be zero).

Grouping Characters (68020–plus and CPU32 only)

[ ] enclose an indirect expression and are **required** characters.

( ) enclose the entire *<ea>* expression and are **required** characters.

Table 2–1 summarizes the addressing modes defined for the M68000 family, their invocations, and significant constraints.

Table 2–1: Address Modes

MODE	INVOCATION	COMMENTS
1. Register Direct	An Dn	
2. Memory Address		
a) Simple indirect	(An)	
b) Predecrement	–(An)	
c) Postincrement	(An)+	
d) Indirect with displacement	<absolute>(An) <simple>(An) <complex>(An)	Displacement must fit in 16–bit signed integer.
e) Indirect with index	<absolute>(An,Ri)	Displacement must fit in 8–bit signed integer.
f) Indirect with preindexing plus base and outer displacements (68020–plus/CPU32 only)	([bd,An,Ri],od)	Inner and outer displacement may be 8, 16, or 32 bits Brackets([]) are required.
g) Indirect with postindexing plus base and outer displacements (68020–plus/CPU32 only)	([bd,An],Ri,od)	Inner and outer displacement may be 8, 16, or 32 bits. Brackets([]) are required.
h) Direct with indexing plus base displacement (68020–plus/CPU32 only)	(bd,An,Ri)	Displacement may be 8, 16, or 32 bits.

MODE	INVOCATION	COMMENTS
3. PC–relative Address		
a) PC with 16–bit displacement	<simple>	Expression is an address (not a displacement) which must be backward, within current relocatable section.
	<absolute>(PC) <simple>(PC) <complex>(PC)	Forced PC–Relative. Computed relative displacement must fit within 16–bit signed integer. Resolved at assembly or link time.
b) PC with index (16– or 32–bit) plus 8–bit displacement	<simple>(Ri)	Expression is an address which must be backward, and within current relocatable section. Ri must not be just An.
	<absolute>(PC,Ri)	Forced PC–Relative.
	<simple>(PC,Ri)	Expression must be within current program section.
c) PC with preindexing plus base and outer displacements. (CPU32/68020–plus only)	([bd,PC,Ri],od)	Brackets ([]) are required.
d) PC with postindexing plus base and outer displacements. (CPU32/68020–plus only)	([bd,PC],Ri],od)	Brackets ([]) are required.
e) PC indirect with indexing plus base (CPU32/68020–plus only)	(bd,PC,Ri)	
f) Implicit PC reference		Implied by branch instructions (BSR, Bcc

MODE	INVOCATION	COMMENTS
		or SBcc). The computed relative displacement is either 8 (not DBcc), 16 or 32 bits (CPU32/68020–plus only) depending on assembler options (BRS,BRB, BRW, and BRL), and whether these options are overridden on the current instruction, e.g., BSR.S.
4. Direct Address		
a) Absolute (16– or 32–bit)	<absolute> <simple> <complex> (<absolute>). (<absolute>).L	Expression must be forward reference or not in current program section. Forced short. Forced long.
5. Immediate		
a) Immediate (8–, 16–, or 32–bit)	#<absolute> #<simple> #<complex>	

Table 2–2 provides a cross reference of operand formats and addressing modes. Given an operand of the format shown in the first column, the other columns show which addressing mode is indicated, depending on whether the expression is absolute, simple relocatable, or complex relocatable.

Table 2–2: Effective Addressing Mode, Given Operand Format and <expr> Type

Effective Addressing Mode			
Operand Format	**Absolute <expr> <expr>**	**Simple Relocatable <expr>**	**Complex Relocatable**
<expr>(An)	d(An)	d(An)	d(An)
<expr>(Dn)	invalid	d(PC,Dn) *	invalid
<expr>(An,Ri)	d(An,Ri)	invalid	invalid
<expr>	absolute(W,L)	d(PC) or absolute(W,L)	absolute(W,L)
<expr>(PC)	d(PC)	d(PC)	d(PC)
<expr>(PC,Ri)	d(PC,Ri) *	d(PC,Ri) *	invalid
#<expr>	immediate(B,W,L)	immediate(W,L)	immediate(W,L)
	* Must be within current program section.		

4.9.1 Register Direct Modes

These effective addressing modes specify that the operand is in one of the 16 general purpose registers (eight data and eight address registers). The operation is performed directly on the actual contents of the register.

Syntax

A*n* or **D***n* where *n* is between 0 and 7.

Example

```
        CLR.L     D1        Clear all 32 bits of
*                           D1.
*
        ADD       A1,A2     Add low order word of
*                           A1 to low order word
*                           of A2.
```

Memory Address

The following effective addressing modes specify that the operand is in the address register specified by the register field.

Address Register Indirect

The address of the operand is in the address register specified by the register field.

Syntax

(**A***n*) where *n* is between 0 and 7.

Example

```
        MOVE      5,(A5)    Move value 5 to word
*                           whose address is
*                           contained in A5.
*
        SUB.L     (A1),D0   Subtract from D0
*                           the value in the
*                           long word whose
*                           address is contained
*                           in A1.
```

Address Register Indirect with Postincrement

The address of the operand is in the address register specified by the register field. After the operand address is used, it is incremented by one, two, or four, depending upon whether size of the operand is byte (.B), word (.W), or long (.L).

Syntax

(A*n*)+ where *n* is between 0 and 7.

Example

```
        MOVE.B    (A2)+,D5     Move byte whose
*                              Address is in A2 to
*                              low order byte of D2:
*                              increment A2 by 1.
*
        SUB.L     (A1)+,D0     Subtract from D0
*                              the value in the long
*                              word whose address is
*                              contained in A1;
*                              Increment A1 by 4.
```

Address Register Indirect with Predecrement

The address of the operand is in the address register specified by the register field. Before the operand address is used, it is decremented by one, two, or four, depending upon whether the operand size is byte (.B), word (.W), or long (.L).

Syntax

–(A*n*) where *n* is between 0 and 7.

Example

```
        CLR       -(A2)          Subtract 2 from A2;
*                                Clear word whose
*                                address is now given
*                                by A2.
*
        CMP.L     -(A1),D0       Subtract 4 from A1;
*                                Compare longword
*                                whose address is now
*                                in A1 with
*                                contents of D0.
```

Address Register Indirect with Displacement

The address of the operand is the sum of the address in the address register and the sign extended displacement.

Syntax

d(A*n*) where *n* is between 0 and 7.

Example

```
DLC     EQU       32             (For Use in the Next
*                                Instruction)
        CLR.B     DLC(A2)        Clear byte whose
*                                address is given by
*                                adding 32 to the
*                                contents of A2.
        MOVE      #2,10(A1)      Move value 2 to the
*                                word whose address
*                                is given by adding
*                                10 to the contents
*                                of A1.
```

Address Register Indirect with Index

The address of the operand is the sum of the address in the address register, the sign extended displacement, and the contents of the index (A or D) register. On the 68020–plus and CPU32, the index register may be multiplied by a scale factor of 2, 4, or 8.

Syntax

d(**A***n*,R*n*[.**W**][**scl*])
Specifies low order word of index register R*n*.

d(**A***n*,R*n*.**L**[**scl*])
Specifies entire contents of index register R*n*.

Note: The scale factor *scl is only permitted for 68020–plus and CPU32.

Example

```
        ADD         AVAL(A1,D2),D5
* Add to low order word of D5 the word whose
* address is given by adding the contents of
* A1, the contents of the low order word of
* D2, and the displacement denoted
* symbolically by AVAL.

        MOVE.L      D5,$20(A2,A3.L)
* Move entire contents of D5 to the longword
* whose address is given by adding the
* contents of A2, the contents of A3, and
* the displacement of hex 20.
```

Address Register Indirect with Preindexing Plus Base and Outer Displacements (68020–plus and CPU32)

The address of the operand is the sum of the <iea> and a sign extended outer displacement value od. <iea> is the sum of the contents of the address register An (or ZAn), the base displacement bd, and the contents of the index register Rn (or ZRn). The index register may be multiplied by a scale factor of 2, 4, or 8. Therefore,

```
bd + (An) + (Rn)*scl ---> <iea>
(<iea>) + od ---> <operand>
```

Syntax

([*bd*,A*n*,R*n*[.W][**scl*]],*od*)
Specifies low order word of index register R*n*.

([*bd*,A*n*,R*n*.L[**scl*]],*od*)
Specifies entire contents of index register R*n*.

Example

```
        OR          ([BASE,A1,D2*4],AVAL),D5
* Form an indirect effective address <iea> by
* adding the value of BASE, the contents of A1,
* and the contents of the low order word of D2
* times 4. Add the 32-bit word located at
* <iea> to the value of AVAL to form an
* effective address <ea>. OR the word at <ea>
* into the low order word of D5.
*
        AND         ([2,D2,A1],4),D5
* In this example, the assembler selects the
* leftmost address register (A1) to be the
* base register.
```

Address Register Indirect with Postindexing Plus Base and Outer Displacements (68020–plus and CPU32)

The address of the operand is the sum of the <iea>, the contents of the index register Rn (or ZRn), and the outer displacement value od. The index register may be multiplied by a scale factor of 2, 4, or 8. <iea> is the sum of the base displacement bd and the contents of the base register An (or ZAn). Therefore,

```
bd + (An) ---> <iea>
(<iea>) + od + (Rn)*scl ---> <operand>
```

Syntax

([*bd*,**A***n*],*od*,R*n*[.**W**][**scl*])
Specifies low order word of index register R*n*.

([*bd*,**A***n*],*od*,R*n*.**L**[**scl*])
Specifies entire contents of index register R*n*.

Example

```
        ADD         ([BASE,A1],AVAL,D2*2),D5
* For an indirect effective address <iea> by
* adding the value of BASE and the contents
* of register A1. The contents of the
* resultant address <iea> are added to the
* value of AVAL and the low order word of D2
* times two to form the address of the operand
* to be added to the contents of D5.
```

Address Register Indirect with Indexing Plus Base Displacement (68020–plus and CPU32)

The address of the operand is the sum of the 8–bit base displacement bd, the contents of the base register An, and the

contents of the index register Rn. The index register may be multiplied by a scale factor of 2, 4, or 8. Therefore,

```
bd + (An) + (Rn)*scl ---> <operand>
```

Syntax

(*bd*,A*n*,R*n*[.W][**scl*])
Specifies low order word of index register R*n*.

(*bd*,A*n*,R*n*.L[**scl*])
Specifies entire contents of index register R*n*.

Example

```
        ADD          (48,A1,D2),D5
* The sum of the constant 48, the contents of
* base register A1, and the low order word of
* base register D2 form the address of the
* operand to be added to D5.
*
        ADD          (48000,D1,A2),D5
* In this example the displacement is 48000.
* D1 is interpreted as the index register
* even though it comes before A2.
```

Program Counter with Displacement

The address of the operand is the sum of the address in the program counter and the sign extended displacement integer. The assembler calculates this sign extended displacement by subtracting the address of displacement word from the value of the operand field.

Syntax

<expression>(PC)

Forced program counter–relative. Note that *<expression>* is interpreted as a program address rather than a displacement.

Example

```
        MOVE        TAG(PC),A0
* Move from TAG using program counter-relative
* addressing. Note that the assembler would do
* this without the (PC) if TAG were a backwards
* reference to the same section.
```

Program Counter with Index

The address is the sum of the address in the program counter, the sign extended displacement value, and the contents of the index (A or D) register. On the 68020–plus and CPU32, the index register may be multiplied by a scale factor of 2, 4, or 8.

Syntax

<*expression*>(R*n*[.**W**][**scl*])
Specifies the low order word of index register R*n*.

<*expression*>(R*n*.**L**[**scl*])
Specifies the entire contents of index register R*n*.

<*expression*>(**PC**,R*n*[.**W**][**scl*])
Forced program counter–relative.

<*expression*>(**PC**,R*n*.**L**[**scl*])
Forced program counter–relative.

Note: <expression> is interpreted as a program address rather than a displacement.

Example

```
        MOVE.L      TABLE(D2),D4
*  Move the longword at location (TABLE plus
*  contents of D2) to D4. TABLE must be a
*  backwards reference not more than -126
*  bytes before the start of the instruction.
        JMP         TABLE(PC,A2.W)
*  Transfer control to location defined by TABLE
*  plus the low order word of A2 (with sign
*  extension). TABLE will be addressed using
*  PC-relative addressing. TABLE must be within
*  the address range of -126 before the start
*  of the instruction to +129 bytes after the
*  instruction.
```

Program Counter with Preindexing Plus Base and Outer Displacements (68020–plus and CPU32)

The address of the operand is the sum of the <iea> and a sign extended outer displacement value od. <iea> is the sum of the contents of the Program Counter PC (or ZPC), the base displacement bd, and the contents of the index register Rn (or ZRn). The index register may be multiplied by a scale factor of 2, 4, or 8. Therefore,

```
bd + (PC) + (Rn)*scl ---> <iea>
(<iea>) + od ---> <operand>
```

Note: Whenever ZPC is used, bd is not offset by the current PC value.
od is never offset by the PC value.

Syntax

([*bd*,**PC**,R*n*[.**W**][**scl*]],*od*)
Specifies the low order word of index register R*n*.

([*bd*,**PC**,R*n*.**LI**[**scl*]],*od*)
Specifies the entire contents of index register R*n*.

Example

```
        MOVE.L      ([BASE,PC,A2],AVAL),D5
* Form an indirect effective address <iea> by
* adding the contents of the low order word of
* A2 to the address of BASE. The contents of
* the resulting address, <iea>, are added to
* the value of AVAL to give the address of
* the value to be moved to D5. The instruction
* contains the displacement between BASE and
* the address of the displacement field within
* the MOVE instruction.
```

Program Counter with Postindexing Plus Base and Outer Displacements (68020–plus and CPU32)

The address of the operand is the sum of the <iea>, the contents of the index register Rn (or ZRn), and the outer displacement value od. The index register may be multiplied by a scale factor of 2, 4, or 8. <iea> is the sum of the base displacement bd and the contents of the program counter PC (or ZPC). Therefore,

```
bd + (PC) ---> <iea>
(<iea>) + od + (Rn)*scl ---> (operand)
```

Note: Whenever ZPC is used, bd is not offset by the current PC value. od is never offset by the PC value.

Syntax

([*bd*,**PC**],*od*,R*n*[.**W**][**scl*])
Specifies the low order word of index register R*n*.

([*bd*,**PC**],*od*,R*n*.**L**[**scl*])
Specifies the entire contents of the index register R*n*.

Example

```
        ADD          ([BASE,PC],AVAL,D2),D5
* The address of BASE is the indirect effective
* address <iea>. The contents of the longword
* at <iea> is added to the value of AVAL and
* the low order word of D2 to form the address
* of the operand to be added to D5.
```

Program Counter Indirect with Indexing Plus Base Displacement (68020–plus and CPU32)

The address of the operand is the sum of the sign extended base displacement bd, the contents of the program counter PC, and the contents of the index register Rn. The index register may be multiplied by a scale factor of 2, 4, or 8. Therefore,

```
bd + (PC) + (Rn)*scl ---> <operand>
```

Syntax

(*bd*,**PC**,R*n*[.**W**][**scl*])
Specifies the low order word of index register R*n*.

(*bd*,**PC**,R*n*.**L**[**scl*])
Specifies the entire contents of the index register R*n*.

Example

```
        ADD          (BASE,PC,D2),D5
* The sum of the address of BASE and the
* contents of the low order word of D2 forms
* the address of the operand to be added to
* D5. The instruction contains the
* displacement between BASE and the address
* of the displacement field within the ADD
* instruction.
```

Direct Address Modes

Direct address modes use the effective address register field to specify the direct addressing mode instead of a register number. Table 2–3 provides the ranges for absolute short and long addresses.

Table 2–3: Special Address Ranges

32–Bit Address	16–Bit Representation of 32–Bit Address	
000000 . . 00007FFF	0000 . . 7FFF	Absolute short
00008000 . . FFFF7FFF		(No representation in 16 bits;must be absolute long)
FFFF8000 . . FFFFFFFF	8000 . . FFFF	Absolute short

Absolute Short Address

The 16–bit address of the operand is sign extended before it is used. Therefore, the useful address range is 0 through $7FFF and $FFFF8000 through $FFFFFFFF.

Syntax

XXX

Example

```
JMP      $400      Jump to hex address 400
```

An absolute short address can be forced with the notation:

```
(XXX).W
```

Absolute Long Address

The address of the operand is the 32–bit value specified.

Syntax

XXX

Example

```
JMP      $12000      Jump to hex address 12000
```

An absolute long address can be forced with the notation:

```
(XXX).L
```

4.9.2 Immediate Data

An absolute number may be specified as an operand by immediately preceding a number or expression with an immediate character. The immediate character (#) is used to designate an absolute number other than a displacement or an absolute address.

Syntax

#XXX

Example

```
SUB.L    #1,D0    Subtract 1 from D0.
```

4.9.3 Notes on 68020–plus and CPU32 Addressing Modes

The 68020–plus and CPU32 processors support several additional addressing modes. The relation between the 68000/68010, 68020–plus, and CPU32 addressing modes are discussed in the following sections and are summarized in Table 2–4. The mode numbers mentioned in the following sections refer to the **mode** and **register** values specified in Table 2–3 of the *MC68030 32–Bit Microprocessor User's Manual.*

Address Register Addressing Modes

One of the major changes to the addressing modes in the 68020–plus and CPU32 is in the mode 6 <ea> expressions, especially the "address register indirect with indexing plus base displacement" addressing mode. Some source code variations of this new 68020–plus and CPU32 mode are redundant with 68000 addressing modes 2, 5, and 6, in the sense that the final effective address is the same.

When a redundant mode occurs, the assembler will select the more efficient form whenever possible. For example, when the assembler sees the form:

```
(An)
```

it will generate addressing mode 2 (address register indirect). Similarly, the assembler will generate addressing mode 5 (address register indirect with displacement) when it sees the following two forms:

```
d(An)          or the new syntax form
(bd,An)        when bd fits in 16 bits or less
```

Similar remarks apply to the 68000 mode 6 addresses:

```
(An,Ri)
d(An,Ri)       or the new syntax form
(bd,An,Ri)     when bd fits in 8 bits or less
```

Of course, 68000 modes cannot be generated if a scaling factor other than 1 is present.

There are cases when the assembler cannot know for certain which addressing mode is best. For example, if a displacement is specified using an external, the assembler may not know whether or not it will fit in an 8 or 16 bit field. In these cases the assembler will take the most conservative viewpoint available to it. This depends upon which target (68000/68010 or 68020–plus and CPU32) the assembler is being used for.

If the assembler is used for a 68000, then the appropriate 68000 addressing mode and an error is issued only if the value cannot fit in the required range. If the assembler is used for a 68020–plus or CPU32, then generally the addressing mode which is certain to work will be used, even if it is less efficient. The only exception to this rule is that the assembler may generate the more efficient (but more restricted) addressing mode when the older syntax is used. However, it will NOT do this if the assembler knows that the value will NOT fit in the required field. For example, the d(An) form will generate addressing mode 5 if the assembler does not know whether or not d will fit in a 16 bit field.

The programmer can force the assembler to generate the less efficient mode 6 instructions by using the suppressed registers. This is described in detail in the *Using Suppressed Registers to Force Redundant Addressing Modes* section.

Program Counter Relative Addressing Modes

Similar considerations apply to the PC–relative addressing modes as were described in the previous section. For example, when the assembler sees:

```
bd(PC)      or the new syntax form
(bd,PC)     when bd fits in 16 bits or less
```

it generates the more efficient 68000 PC indirect relative addressing mode (mode = 111 and register = 010, or mode 72) rather than the 68020–plus and CPU32 PC indirect with displacement addressing mode (mode = 111 and register = 011, or mode 73). The programmer can force the assembler to generate the redundant mode 73 instructions by using suppressed registers.

All mode 73 <ea> expressions require PC or ZPC as part of the expression to distinguish them from their address register counterparts. All mode 72 and 73 references are to program space and all mode 2, 5, and 6 references are to data space. When the program counter is suppressed (ZPC), the displacement is assumed to be absolute and hence is not offset from the current PC value.

Using Suppressed Registers to Force Redundant Addressing Modes

Register mnemonics ZPC, ZA0–ZA7 and ZD0–ZD7 imply registers whose values are always taken to be zero. These symbols may be used to specify any allowable register while at the same time suppressing that register during <ea> calculations. These symbols are included for diagnostic purposes so that every field of the object code instruction can be specified. It also indicates whether the PC or An is being suppressed, and this determines whether the <ea> is in instruction space or data space. By default, ZA0 is taken to be the suppressed register if no register is specified. ZPC must appear in the <ea> expression to force PC–relative addressing with PC suppressed.

Where an <ea> expression would normally default to a 68000 addressing mode, the equivalent <ea> may be forced to mode 6 or 73 by including ZA0 within the <ea> expression. This is because the assembler always selects the most efficient addressing mode unless another equivalent mode is forced.

When ZA0 follows the closing square bracket, i.e., ([<ea>],ZA0), it forces postindexed indirect modes where the index register has been suppressed.

Registers can be suppressed only in the Address Register Indirect with Base Displacement and the Program Counter Indirect modes.

CPU32 Addressing Mode

The CPU32 addressing mode is the same as the 68020–plus, except that the memory indirect modes are absent.

Addressing Summary

Table 2–4 summarizes much of the information presented in the preceding sections:

Table 2–4: Addressing Summary

Default Addressing Mode		Functionally Equivalent Forced Addressing Modes	
Syntax	**Mode**	**Syntax**	**Mode**
0	Mode 70	(ZRi)	Mode 6n w/null bd
0	Mode 70	((0).W,ZRi)	Mode 6n w/16–bit bd=0
0	Mode 70	((0).L,ZRi)	Mode 6n w/32–bit bd=0
bd		(bd,ZRi)	Mode 6n
(An)	Mode 2n	(An,ZRi)	Mode 6n
bd(An)	Mode 5n	(bd,An,ZRi)	Mode 6n
none		(ZPC)	Mode 73 w/null bd
none		((0).W,ZPC)	Mode 73 w/16–bit bd=0
none		((0).L,ZPC)	Mode 73 w/32–bit bd=0
bd		(bd,ZPC)	Mode 73
([])	Mode 6n w/null bd	([(0).W])	Mode 6n w/16–bit bd=0
([])	Mode 6n w/null bd	([(0).L])	Mode 6n w/32–bit bd=0
([])	Mode 6n w/preindex	([],ZRi)	Mode 6n w/postindex
([])	Mode 6n w/supp. An	([ZPC])	Mode 73 w/supp. PC
Note: "n" mode numbers refer to the base address register.			

4.9.4 Notes on Addressing Options

By default, the assembler resolves all forward references by using the longer form of the effective address in the operand reference. The programmer may override this default by specifying OPT FRS (or the command line option), which designates that forward absolute references should be short, or OPT BRB (or the command line option), designating that forward relative branches should use the shorter (8–bit) displacement format. For the 68020–plus and CPU32, OPT BRW (or the command line option) can be used to force 16–bit (rather than 32–bit) displacements on forward branches.

On an instruction which does not allow a size code, the current forward reference default format may be overridden (for that instruction only) by appending .S (short) or .L (long) to the instruction mnemonic. A similar override may be performed in the structured syntax control directives via the extent codes (see the *Syntax* section of the *Structured Control Statements* chapter in this manual for further explanation). No override is possible on instructions with size code specification. Notably, this override procedure is possible on the JMP and JSR instructions.

The shorter form of the effective address for relative branch instructions is an 8–bit displacement; the longer format is a 16–bit displacement. For absolute jumps, the shorter effective address is the 16–bit absolute short; the longer format is the 32–bit absolute long mode.

In the case of forward references in either relative branches or absolute jumps, if the shorter format is directed and the longer format is later found necessary when the reference is resolved, an error will occur.

References to symbols already defined, whether absolute or relative, are resolved by the assembler into the appropriate effective address, unless .S or .L is forced on the instruction.

A short form may be forced by following the instruction mnemonic with .S.

Example

```
        BEQ.S     LOOP1          If condition code EQ
*                                (equal) is true, then
*                                to LOOP1 (using the
*                                short form of the
*                                instruction).
```

In this case, the instruction size is forced to one word. An error will be printed if the operand field is not in the range of an 8–bit displacement.

Unlike the Motorola assembler, the InterTools assembler permits relocation of 8–bit values. If, however, the symbol happens to reside beyond the 8–bit relocation after the program has been located, the error will not be detected until the object module is formatted into its hex or binary image.

Default actions of the assembler have been chosen to minimize two common address mode errors:

- *Displacement range violations*:
  Relative branch instructions (Bcc, BRA, BSR) allow either 8–bit or 16–bit displacements form the PC. On forward references in such instructions, the default action is to assume the 16–bit displacement.
- *Inappropriate absolute short address*:
  Absolute addresses may be short (16–bit) or long (32–bit). On forward references with absolute effective addresses, the default action is to assume the long format. The long form is also assumed on references to another section (unless it is a SECTION.S).

Default conditions have been chosen to prevent errors by using addressing formats which ensure address resolution in the broadest range of conditions, even at the expense of code efficiency. Each default may be overridden to improve efficiency or to create position–independent code. Also, the current address size defaults may be overridden in certain cases on specific instructions which do not allow size codes by appending .S or .L, as in JMP.S and JMP.L (JMP and JSR only).

The previous discussion assumed relative branches could not be 32 bits. This is not the case when using the 68020–plus or CPU32.

The resolution of operands into effective address modes (ignoring base register addressing) is summarized in the Tables 2–5 through 2–10.

Table 2–5: Operand Resolution

	INSTRUCTION FOLLOWS:	
OPERAND TYPE	**SECTION**	**ORG**
Known location (backward in pass 1)	See Table 2–6	See Table 2–7
Unknown location (forward)	See Table 2–8	See Table 2–8
External reference	See Table 2–9	See Table 2–10

Table 2–6: Known* Location of Operand and Instruction Follows Section

Operand Reference	Option In Effect When Instruction Occurred	Effective Address Mode
Simple relocation	PCS	PC relative resolved by linker if operand & instructions are in different SECTIONs. IF displacement > 16–bit THEN error
	NOPCS (default)	IF operand and instruction in same SECTION and displacement <= 16–bit THEN PC relative ELSE IF operand defined in SECTION.S THEN absolute short ELSE absolute long (resolved by linker)
Complex relocation	(Any)	Absolute long
Absolute (ORG)	(Any)	Absolute short or absolute long depending on the value of the operand

* Label defined before instruction which references it (in pass 1).

Table 2–7: Known* Location of Operand and Instruction Follows

Operand Reference	Option in Effect When Instruction Occurred	Effective Address Mode
Simple relocation	(Any)	IF operand defined in SECTION.S THEN absolute short ELSE absolute long (resolved by linker)
Complex relocation	(Any)	Absolute long
Absolute (ORG)	PCO	IF displacement <= 16–bit THEN PC relative ELSE absolute short or absolute long depending on value of operand
	NOPCO (default)	Absolute short or absolute long depending on the value of the operand

*Label defined before instruction which references it (in pass 1).

Table 2–8: Unknown** Location of Operand and Instruction Follows SECTION or ORG

Operand Reference	Option In Effect When Instruction Occurred	Effective Address Mode
(All)	FRS	Absolute short (resolved by linker)
	FRL (default)	Absolute long (resolved by linker)

** Label undefined at time of reference (error at pass 2).

Table 2–9: External Reference and Instruction Follows SECTION

Operand Reference	Option in Effect When Instruction Occurred	Effective Address Mode
XREF with SECTION designation	PCS	PC relative (resolved by linker)
	NOPCS	IF operand defined in
Example	(default)	SECTION.S or XREF.S
XREF 2:L1		THEN absolute short
		ELSE absolute long
		(resolved by linker)
XREF without SECTION designation	(Any)	IF operand defined with XREF.S
		THEN absolute short
		ELSE (see below)
Example		(resolved by linker)
XREF L1		
	FRS	Absolute short
		(resolved by linker)
	FRL	Absolute long
	(default)	(resolved by linker)

Table 2–10: External Reference and Instruction Follows ORG

Operand Reference	Option in Effect When Instruction Occurred	Effective Address Mode
XREF with SECTION designation Example XREF 2:L1	(Any)	If operand defined in SECTION.S or XREF.S THEN absolute short ELSE absolute long (resolved by linker)
XREF without SECTION designation Example XREF L1	(Any)	IF operand defined with XREF.S THEN absolute short ELSE (see below)
	FRS	Absolute short (resolved by linker)
	FRL (default)	Absolute long (resolved by linker)

4.10 Instruction Mnemonics

The instruction operations described in this section are used by the assembler for the 68000, 68010, 68332, 68020, 68030, 68040, 68060 68EC030, and 68EC040. The 68000 family is upwards compatible, which means that 68000 instructions will run on the 68010; 68010 instructions will run on the 68332 or 68020; 68020 instructions will run on the 68030; and 68030 instructions will run on 68040, 68060 68EC030, or 68EC040. Unless where indicated otherwise, all the instructions described in this section are 68000 instructions.

The 68020–plus supports a general purpose coprocessor interface, which allows instructions for a class of coprocessors to be routed to the coprocessor by the 68020–plus CPU. The following coprocessors are supported by the InterTools Assembler at this time: the MC68881/68882 floating point coprocessor and the 68851 paged memory management unit coprocessor. Instructions for these coprocessors are described in the *MC68881/68882 Floating Point Coprocessor Instructions and Floating Point Instructions supported by the MC68040* section and the *MC68851 Paged Memory Management Unit Coprocessor Instructions* section.

4.10.1 Arithmetic Operations

Syntax

<operation>[.*<size>*] *<source>*,*<destination>*

where:

<operation> = ADD|SUB|MULS|MULU|DIVS|DIVU
<size> = B|W|L (W is default)

The arithmetic operations include add, subtract, multiply, and divide. Add and subtract are available for all data operand sizes. Multiply and divide may be signed or unsigned, but do not allow byte size operands. Operations on decimal data (not shown) include add, subtract, and negate.

Example

```
        ADD.W    D1,D2          Adds low order word
*                               of D1 to low order
*                               word of D2.
        SUB.B    #5,(A1)        Subtracts value 5
*                               from byte whose
*                               address is contained
*                               in A1.
```

The word forms of the multiply instructions multiply two 16–bit operands yielding a 32–bit product. These are the only forms supported by the 68000/68010. The 68020–plus and CPU32 multiply instructions have been extended to support larger operands. There are two new forms:

- 32*32 => 64
- 32*32 => 32

These two forms are distinguished by the syntax of the operands. In the first form, two data registers, Dh:Dl, are coded. One data register, Dl, holds the multiplier before multiplication and the low order longword of the product after multiplication. Another data register, Dh, holds the high order longword of the product after multiplication (Dh must be different from Dl). In the second form only the one register, Dl, is specified.

Syntax

MULS.L *<ea>*,[**D***h*:]**D***l* (*signed multiply*)
MULU.L *<ea>*,[**D***h*:]**D***l* (*unsigned multiply*)

The word forms of the divide instructions divide a 32–bit dividend by a 16–bit divisor. Before the division the destination operand contains the dividend. After the division it contains a 32–bit value with the

quotient in the low order 16 bits and the remainder in the high order 16 bits. These are the only forms supported by the 68000/68010. The 68020–plus and CPU32 divide instructions have been expanded to support larger operands. There are three new forms:

- 64/32 => 32r:32q
- 32/32 => 32q
- 32/32 => 32r:32q

The first two forms are specified by coding the .L size suffix on the divide mnemonic (DIVS.L or DIVU.L). These forms are then distinguished by the syntax of the operands. In the first form, two data registers, Dr:Dq, are coded. Here, Dr holds the high order longword of the dividend before division and the remainder after division, and Dq holds the low order longword of the dividend before division and the quotient after division.

In the second form, a single data register Dq is specified (or Dr equals Dq). Then Dq is used as the entire dividend for the divide, and contains the 32–bit quotient after the division. No remainder is returned in this case.

The third form is specified by using a different instruction mnemonic, e.g., DIVSL rather than DIVS, in addition to the .L suffix. In this case, two data registers, Dr:Dq are required. Dq is used as the entire dividend for the divide, and contains the 32–bit quotient after the division. The remainder is returned in Dr.

Syntax

DIVS.L <*ea*>,[**D***r*:]**D***q* (*signed division*)
DIVSL.L <*ea*>,**D***r*:**D***q* (*signed division*)
DIVU.L <ea>,[**D***r*:]**D***q* (*unsigned division*)
DIVUL.L <*ea*>,**D***r*:**D***q* (*unsigned division*)

Example

```
      MULS.L   D1,D3
*  Multiply D1 by D3. Leave product in D3.
      MULS.L   D1,D2:D3
*  Multiply D1 by D3.
*  Leave high half of 64-bit product in D2;
*  low half in D3.
      DIVS.L   D1,D3
*  Divide D1 into D3. Leave quotient in D3
*  (no remainder).
      DIVSL.L  D1,D2:D3
*  Divide D1 into D3. Leave quotient in D3
*  and remainder in D2.
      DIVS.L   D1,D2:D3
*  Divide D1 into the 64-bit integer whose
*  high half is in D2 and whose low half is
*  in D3. Leave the quotient in D3 and the
*  remainder in D2.
```

4.10.2 Move Instruction

Syntax

MOVE[.<*size*>] <*source*>,<*destination*>

where:

<*size*> = B | W | L (W is default)

The MOVE instruction is used to move data between registers and/or memory.

Example

```
        MOVE     D1,D2          Moves low order word
*                               of D1 into low
*                               order word of D2.

        MOVE.L   XYZ,DEF        Moves longword
*                               addressed by XYZ into
*                               longword addressed
*                               by DEF.

        MOVE.W   #A,ABC         Moves word with value
*                               of $4100 into word
*                               addressed by ABC.

        MOVE     ADDR,A3        Moves word addressed
*                               by ADDR into low
*                               order word of A3.
```

4.10.3 Compare and Check Instructions

Syntax

CMP[.<*size*>] <*operand1*>,<*operand2*>
TST[.<*size*>] <*operand*>
CHK <*ea*>,**D***n*
CHK.L <*ea*>,**D***n* (*68020–plus/CPU32 only*)
TAS <*operand*>

where:
<*size*> = B | W | L (W is default)

CMP sets the condition codes by subtracting <*operand1*> from <*operand2*> without altering <*operand1*> or <*operand2*>. If **D***n* is less than 0 or greater than the value at <ea>, then CHK initiates a system trap.

Example

```
        CMP.L     ADDR,D1       Compare longword at
*                               location ADDR with
*                               contents of D1,
*                               setting condition
*                               codes accordingly
        CHK       (A0),D3       Compare word whose
*                               address is in A0
*                               with low order word
*                               of D3. Initiates a
*                               trap if check fails.
```

Condition codes resulting from the execution of the compare instruction are set so that a "less than" condition means that operand2 is less than operand1, and "greater than" means that operand2 is greater than operand1.

There are actually four compare instructions: CMP, CMPA, CMPI, and CMPM. CMPI requires an immediate as the first operand, CMPA requires an address register as the second operand, CMP requires a data register as the second operand, and CMPM requires both operand to use the postincrement addressing mode, "(An)+". The assembler will choose the appropriate mnemonic if CMP is coded. Note that the CMPI instruction only allows PC–relative addressing for the second operand on the 68020–plus and CPU32.

The TST instruction is equivalent to comparing with zero.

The CHK instruction will cause a system trap if the register contents are less than zero or greater than the value specified by the source operand. The vector number if generated to reference the CHK instruction exception vector.

The TAS (test and set) operation is a synchronization primitive. It simultaneously tests the operand and sets its high order bit. Only byte size operands are permitted.

4.10.4 Logical Operations

Syntax

<*operation*>[.<*size*>] <*source*>,<*destination*>
<*operation*>[.<*size*>] #<*immediate*>,**SR**
<*operation*>[.<*size*>] #<*immediate*>,**CCR**
NOT[.<*size*>] <*operand*>

where:
<*operation*> = AND | OR | EOR
<*size*> = B | W | L (W is default)

Logical operations include AND, OR, EXCLUSIVE OR, and NOT. The instructions accept register, memory, or immediate operands, except that the destination operand may not be an immediate. If the first operand is an immediate then the destination may also be the status register (SR) or the condition code register (CCR).

Example

```
        AND         D1,D2
*  Compute the logical and of the low order
*  words of D1 and D2, and store the result
*  in the low order word of D2.
```

4.10.5 Shift Operations

Syntax

<*operation*>[.<*size*>] <*count*>,**D***n*
<*operation*>[.<*size*>] <*ea*>

where:
<*operation*> = ASL|ASR|LSL|LSR|ROL|ROR|ROXL|ROXR

<size> = B | W | L (W is default)
<count> = **D***n* | immediate
<ea> = *memory reference*

Shift operations include arithmetic and logical shifts, as well as rotate and rotate with extend. All shift operations may be either fixed with the shift count in an immediate field or variable with the count in a register. Shifts in memory of a single–bit position left or right may also be done.

Example

```
        LSL.W    #5,D3          Performs a left,
*                               logical shift of
*                               low order word of D3
*                               by 5 bits; .W is
*                               optional (default).

        ASR      (A2)           Performs a right,
*                               arithmetic shift of
*                               word whose address
*                               is contained in A2;
*                               because this is a
*                               memory operand,
*                               the shift is only
*                               1 bit.

        ROXL.B   D3,D2          Performs a left
*                               rotation with extend
*                               bit of low order
*                               byte of D2;
*                               shift count is
*                               contained in D3.
```

4.10.6 Bit Operations

Bit operations allow test and modify combinations for single bits in either an 8–bit operand for memory destinations or a 32–bit operand for data register destinations. The bit number may be fixed or variable.

Syntax

<operation> *#<bitno>,<operand>*
<operation> **D***n*,*<operand>*

where:
<operation> = BCHG | BCLR | BSET | BTST

Example

```
        BCLR     #3,XY(A3)     Clears bit number 3
*                              in byte whose
*                              address is given by
*                              address in A3 plus
*                              displacement of XY.

        BCHG     D1,D2         Tests a bit in D2,
*                              reflects its value in
*                              condition code Z, and
*                              then changes value of
*                              that bit; bit number
*                              is specified in D1.
```

4.10.7 Set on Condition

Condition codes can be used to set and clear data bytes.

Syntax

S<*cc*> <*location*>

where:
<*cc*> may be one of the following condition codes:

T	CC (or HS)	VC	GE
F	CS (or LO)	VS	LT
HI	NE	PL	GT
LS	EQ	MI	LE

Example

```
        SNE       (A5)          If condition code NE
*                               (not equal) is true,
*                               then set byte whose
*                               address in A5 to 1s;
*                               otherwise, set that
*                               byte to 0.
```

4.10.8 Branch Operations

Branch operations include an unconditional branch, a branch to subroutine, and 14 conditional branch instructions.

Syntax

<*operation*>.<*extent*> <*location*>

where:
<*operation*> = BRA | BSR | B<*cc*>

<*extent*> = B | S | W | L (L is default)
<*cc*> = one of the condition codes described below

	CC (or HS)	VC	GE
	CS (or LO)	VS	LT
HI	NE	PL	GT
LS	EQ	MI	LE

Note that T and F are not valid condition codes for a conditional branch.

Example

```
        BRA     TAG         Unconditional branch
*                           to address TAG.
        BSR     SUBDO       Branch to subroutine
*                           SUBDO.
        BEQ     TRUELAB     Branch to TRUELAB if
*                           EQ condition.
```

All conditional branch instructions use PC–relative addressing only, and may be either one word or two word instructions. On the 68020–plus and CPU32, there is also a three word form. The corresponding displacement ranges are:

- one word: –128...+127 bytes (8–bit displacement)
- two word: –32768...+32767 bytes (16–bit displacement)
- three word: no restriction (32–bit displacement)

Forward references in branch instructions use the longest available format by default. The default may be changed by using assembler

OPT options, or by using the command line switches that mimic the assembler options. For example, OPT BRS or OPT BRB directs the assembler to use the one word form in forward branches, OPT BRW specifies the two word form, and OPT BRL specifies the three word form. The default extent may also be overridden for a single branch operation by appending appropriate extent codes to the instruction. For example:

```
BRA.S    LAB          Short branch to LAB
```

A branch instruction with a byte displacement must not reference the statement which immediately follows it. This would result in an 8–bit displacement value of 0, which is recognized by the assembler as an error condition.

Example (illegal)

```
        BEQ.S    LAB1          LAB1 is the next
LAB1    MOVE     #1,D0         memory word and thus
*                              generates an error.
```

Note: The 68000 and 68010 `.L` extension produces 2–byte offsets where the 68020–plus and CPU32 `.L` extension produces 4–byte offsets.

4.10.9 Jump Operations

Jump operations include a jump to subroutine and an unconditional jump.

Syntax

<*operation*>[.<*extent*>] <*ea*>

where:
<*operation*> = JSR | JMP
<*extent*> = S | L (L is default)

Example

```
        JMP     4(A0)       Unconditional jump to
*                           the location 4 bytes
*                           beyond the address
*                           in A0.

        JMP.L   NEXT        Long (absolute) jump
*                           to the address NEXT.

        JSR     SUBDO       Jump to subroutine
*                           SUBDO.
```

Forward references to a label will use the long absolute address format by default (OPT FRL). The default may be changed to the shorter format by specifying OPT FRS. The default extent may be overridden on a single jump operation to a label by appending S or L as an extent code for the instruction.

4.10.10 Decrement and Branch on Condition (DBcc)

Syntax

DB<*cc*> **D***n*,<*label*>

where:
<*cc*> may be one of the following condition codes:

T	CC (or HS)	VC	GE
F	CS (or LO)	VS	LT
HI	NE	PL	GT
LS	EQ	MI	LE

This instruction is a looping primitive of three parameters: a condition, a data register, and a label. If the specified condition is true then no operation is performed. The condition thus defines a termination condition for the loop. If the termination condition is not true, then the data register is decrement by one. If the register is then –1, execution continues with the next instruction. If the result is not equal to –1, execution continues at the location indicated by <*label*>. <*label*> must be located within a 16–bit displacement from the DB<cc> instruction.

Example

```
        LEA       SOURCE_BUFFER,A0
        LEA       DEST_BUFFER,A1
        MOVE.L    #BYTES_TO_MOVE,D0
LAB     MOVE      (A0)+,(A1)+
        DBF       D0,LAB
```

4.10.11 Move Multiple Registers

Syntax

MOVEM[.<*size*>] <*reg_list*>,<*location*>
(*registers to memory*)
MOVEM[.<size>] <*location*>,<*reg_list*>
(*memory to registers*)

where:
<*size*> = W | L (W is default)
<*reg_list*> is a list of registers or ranges separated by a slash (/).

Register ranges are separated by a hyphen (–). For example, A1–A3 means A1 through A3. When specifying a register range, A and D registers cannot be mixed. For example, A0–A5 is legal, but A0–D0 is not.

This instruction allows the loading and storing of multiple registers. The order in which the registers are processed is independent of the order in which they are specified in the source line; rather, the order of register processing is fixed by the instruction format. For further details, refer to the MOVEM instruction in the *M68000 16/32–Bit Microprocessor Programmer's Reference Manual.*

Example

```
      MOVEM    (A6)+,D1/D5/D7
*  Load registers D1, D5, and D7 from three
*  consecutive words in memory. The address
*  of the first word is given by A6;
*  A6 is increment by 2 after each transfer.

      MOVEM.L  A2-A6,-(A7)
*  Store registers A2 through A6 in five
*  consecutive longwords in memory. The
*  address where A6 is stored is the address
*  in A7; A5 is stored four bytes earlier and
*  so on.
*  A7 is decremented by 4 (because of .L)
*  after each transfer.

      MOVEM    (A7)+,A1-A3/D1-D3
*  Load registers D1, D2, D3, A1, A2, A3
*  (in order) from six consecutive words in
*  memory, starting with the address in A7
*  and incrementing A7 by 2 after each transfer.

      MOVEM.L  A1/A2/A3,REGSAVE
*  Store registers A1,A2, A3 in three
*  consecutive longwords
*  starting with the location labeled REGSAVE.
```

4.10.12 Load Effective Address

This instruction allows computation and loading of the effective address into an address register.

Syntax

LEA *<operand>*,**A***n*

Example

```
      LEA      XYZ(A2,D5),A1
*  Load A1 with the address specified by the
*  first operand. The following two move
*  instructions are then equivalent:
      MOVE     XYZ(A2,D5),D2
      MOVE     (A1),D2
```

4.10.13 Sign Extend Instructions

Syntax

EXT.W **D***n* Byte –> Word
EXTB.L **D***n* Byte –> Long (not 68000/68010)
EXT.L **D***n* Word –> Long

To extend a byte to a word, bit [7] of data register D*n* is copied to bits [15:8] of that register. To extend a byte to a long word, bit [7] of data register D*n* is copied to bits [31:8] of that register. To extend a word to a long word, bit [15] of data register D*n* is copied to bits [31:16] of that register.

Example

```
        EXT       D1              Extend the byte in D1
*                                 to a word
        EXT.W     D1              Extend the word in D1
*                                 to a long word

        EXTB.L    D1              Extend the byte in D1
*                                 to a long word
*  The last instruction is equivalent to the two
*  previous ones.
```

4.10.14 Move to/from Control Register

(68010, 68020–plus, and CPU32 only) With this instruction, the specified control register is copied to the specified general register, or the specified general register is copied to the specified control register. This is always a 32–bit transfer, even though the control register may be implemented with fewer bits. Unimplemented bits read as zeros.

Note: MOVEC is a privileged instruction.

Syntax

MOVEC *<control register>*,*<register>*
MOVEC *<register>*,*<control register>*

where:
<register> = **A***n* | **D***n*
<control register> is one of the registers or pointers listed in the table on the next page:

(68010/68020–plus/ CPU32)	**VBR**	Vector base register
	SFC	Source function code register
	DFC	Destination function code register
	USP	User stack pointer
(68020–plus Only)	**CACR**	Cache control register
	MSP	Master stack pointer
	ISP	Interrupt stack pointer
(68020/68030/68EC030)	**CAAR**	Cache address register
(68040/68060)	**TC**	MMU Translation control register
	ITT0, ITT1, DTT0, DTT1	Transparent translation registers
	URP	User root pointer
(68040)	**MMUSR**	MMU Status register
(68060)	**SRP**	Supervisor root pointer
	BUSCR	Bus control register
	PCR	Processor configuration register
(68EC030)	**AC0, AC1**	Access control registers
	ACUSR	ACU Status register
(68EC040)	**DACR0, DACR1 IACR0, IACR1**	Access control registers
	ACUSR	ACU Status register

Example

```
        MOVEC    VBR,A0        Copies contents of
*                              vector base register
*                              to register A0.

        MOVEC    D7,SFC        Copies contents of
*                              register D7 to the
*                              source function code
*                              register (3 bits).

        MOVEC    DFC,D0        Copies contents of
*                              destination function
*                              code register to
*                              register D0
*                              (3 bits; zero filled).

        MOVEC.L  USP,D7        Copies user stack
*                              pointer to register
*                              D7.
```

4.10.15 Move to/from Address Space

(68020–plus and CPU32 only) Moves a byte, word, or longword from the specified general register to a location within the address space determined by the DFC register, or moves a byte, word, or longword from a location within the address space determined by the SFC register to the specified general register. Note that with a byte operation size specified, the address register direct mode is not allowed.

Note: MOVES is a privileged instruction.

Syntax

MOVES[.<*size*>] <*ea*>,<*register*>
MOVES[.<*size*>] <*register*>,<*ea*>

where:
<*size*> = B | W | L (W is default)

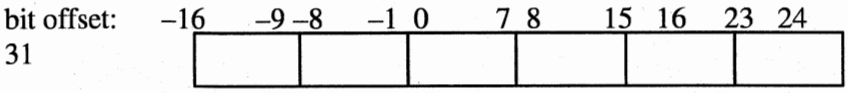
Example

```
        MOVES     (A2)+,D2     Moves a word at the
*                              address contained
*                              in register A2 to
*                              register D2
*                              and then increments
*                              A2 by 2.

        MOVES.L   A4,LABEL     Moves the contents of
*                              register A4 to the
*                              address of LABEL.

        MOVES     2222,A2      Moves one word of
*                              data beginning at
*                              address 2222 to
*                              register A2.
```

<*ea*> must be a data alterable addressing mode. In other words, <*ea*> may not be an immediate, a register, or a PC–relative addressing mode.

4.10.16 Bit Fields and Instructions (68020–plus only)

A bit field is a string of consecutive bits in a bit array. The address of the bit array is determined by the address of the byte containing bit 0 (the base address). Bit number 0 is the most significant bit in the byte at the base address. Bits are numbered from left to right, opposite the order for integers. By this notation, the most significant bit in the next higher byte is bit number 8 and the least significant bit in the next lower byte is bit number –1. Instructions reference bit fields using two parameters: a bit field offset and a bit field width. A bit field offset is the bit field number of the leftmost bit in the field; its range is –2**31 to (2**31) – 1. The bit field width is the number of bits in the bit field; its range is 1 to 32.



byte address: N–2 N–1 N N+1 N+2 N+3
(N = base address)

Single Operand Bit Field Instructions

This section explains the following single operand bit fields: complement, clear, set and test.

Complement Bit Field

Complements a bit field at the specified effective address location. Condition code fields are modified, depending on the value in the bit field before the complement. A bit field is selected by the bit field

offset (the starting bit) and the bit field width (the number of bits included).

Syntax

BFCHG <*ea*>{<*offset*>:<*width*>}

where:
<*offset*> and <*width*> may be an immediate value or a data register D0–D7.

Example

```
      BFCHG    LABEL{0:D1}
*  Complement the bit field located at address
*  LABEL.
*  The bit field extends from bit 0 to bit n-1,
*  where n is the value in D1.
```

Clear Bit Field

Clears a bit field at a specified effective address location. Condition code fields are modified, depending on the value in the bit field before the clear. A bit field is selected by the bit field offset (the starting bit) and the bit field width (the number of bits included).

Syntax

BFCLR <*ea*>{<*offset*>:<*width*>}

where:
<*offset*> and <*width*> may be an immediate value or a data register D0–D7.

Example

```
        BFCLR    LABEL{D1:8}
*   Clear the bit field at address LABEL.
*   The bit field extends from bit n to bit n+7,
*   where n is the value in D1.
```

Set Bit Field

Sets all bits of a bit field at a specified effective address location. Condition code fields are modified, depending on the value in the bit

field before the set. A bit field is selected by the bit field offset (the starting bit) and the bit field width (the number of bits included).

Syntax

BFSET <*ea*>{<*offset*>:<*width*>}

where:
<*offset*> and <*width*> may be an immediate value or a data register D0–D7.

Example

```
      BFSET    22222{0:8}
*  Sets one byte at address 2222 to all 1s.
```

Test Bit Field

Sets condition codes according to the value in the bit field at the specified effective address location. A bit field is selected by the bit field offset (the starting bit) and the bit field width (the number of bits included).

Syntax

BFTST <*ea*>{<*offset*>:<*width*>}

where:
<*offset*> and <*width*> may be an immediate value or a data register D0–D7.

Example

```
      BFTST    (A2){D1:D2}
*  Clears condition codes V and C;
*  sets N and Z according to the bits of the bit
*  field.
```

Double Operand Bit Field Instructions

Extract Bit Field Signed

The bit field at the specified effective address location is sign extended to 32 bits and loaded into a data register. Condition code fields are modified, depending on the value in the bit field before the sign extension. A bit field is selected by the bit field offset (the starting bit) and the bit field width (the number of bits included). The offset and width may be immediate values or a data register.

Syntax

BFEXTS <*ea*>{<*offset*>:<*width*>},**D***n*

Example

```
        BFEXTS   LABEL{0:8},D1
*  Load the value of the byte at LABEL, sign
*  extend it, and store the resulting value in
*  the low order word of D1.
```

Extract Bit Field Unsigned

The bit field at the specified effective address location is zero–extended to 32 bits and loaded into a data register. Condition code fields are modified, depending on the value in the bit field before

the zero extension. A bit field is selected by the bit field offset (the starting bit) and the bit field width (the number of bits included).

Syntax

BFEXTU *<ea>*{*<offset>*:*<width>*},**D***n*

Example

```
      BFEXTU   LABEL{0:8},D1
*  Load the value of the byte at LABEL,
*  zero-extend it, and store the resulting
*  value in the low order word of D1.
```

Find First One in Bit Field

The bit field at the specified effective address location is examined for the most significant bit position that is set. If a set bit exists, the bit offset for that bit is loaded into a data register. If no bit is set, a value is loaded in a data register equal to the offset plus the width of the bit field. Condition code fields are modified, depending on the value in the bit field before the examination. A bit field is selected by the bit field offset (the starting bit) and the bit field width (the number of bits included).

Syntax

BFFFO *<ea>*{*<offset>*:*<width>*},**D***n*

Example

```
      BFFFO    LABEL{0:1},D1
*  If bit 0 in LABEL is set, then set D1 to 0,
*  else set D1 to 1.
```

Insert Bit Field

Move a bit field from the low order bits of the data register to the bit field at the specified effective address location. Condition code fields are modified, depending on the value in the bit field before the insertion. A bit field is selected by the bit field offset (the starting bit) and the bit field width (the number of bits included).

Syntax

BFINS D*n*,<*ea*>{<*offset*>:<*width*>}

Example

```
      BFINS    D1,LABEL{0:D3}
*  Move the low order byte of D1 to the bit
*  field at address LABEL whose width is in
*  register D3.
```

4.10.17 Bounds Check Instructions (68020–plus and CPU32 only)

These instructions compare the value in a register against the lower and upper bound ,pair located at a memory address. When signed comparisons are made, the arithmetically smaller value should be taken as the lower bound. For unsigned comparisons, the logically smaller value should be taken as the lower bound. If the instruction size is byte (.B) or word (.W), only the low order byte or word of a data register, respectively, is used for the comparison. When Rn is an

address register, an instruction size of byte or word results in a sign extension of the bound operands before the comparison.

Syntax

CHK2[.<*size*>] <*ea*>,**R***n*
CMP2[.<*size*>] <*ea*>,**R***n*

where:
<*size*> = B | W | L (W is default)
<*ea*> = address of pair of bounds values; must be a control addressing mode.

The CMP2 instruction sets the condition codes according to the results of the compare. If Rn is out of bounds, then the C bit is set. If Rn is equal to either bound, then the Z bit is set. The CHK2 instruction is similar, except that exception processing is initiated if the register is out of bounds. A vector number that references the CHK instruction exception vector is generated.

Example

```
      CHK2.B   BOUNDS,D1
*  Cause a trap unless the value of the low
*  order byte of D1 is greater than the
*  contents of BOUNDS and less than the
*  contents of BOUNDS+1.
      CMP2     WBOUNDS,D2
*  Sets the C condition code bits if and only
*  if the low order word of D2 is greater than
*  the contents of WBOUNDS and less than the
*  contents of WBOUNDS+2.
```

4.10.18 Packed Decimal (68020–plus only)

These instructions translate data to and from binary coded decimal (BCD) format.

Syntax

PACK –(A*y*),–(A*x*),#<*adjustment*>
PACK D*y*,D*x*,#<*adjustment*>
UNPK –(A*y*),–(A*x*),#<*adjustment*>
UNPK D*y*,D*x*,#<*adjustment*>

Pack BCD

The low four bits of each of two bytes are adjusted and packed into a single byte. When a predecrement addressing mode is specified, two bytes from the source are fetched, adjusted, and concatenated. The extension word is added to the concatenated bytes. Bits [3:0] of each byte are extracted. These eight bits are concatenated to form a new byte which is then written to the destination.

When both operands are data registers, the adjustment is added to the value contained in the source register. Bits [11:8] and [3:0] of the intermediate result are concatenated and placed in bits [7:0] of the destination register.

Unpack BCD

Two BCD digits in the source byte are separated into two bytes with the BCD digit residing in the lower nibble and 0 in the upper nibble. The adjustment is then added to this unpacked value without affecting the condition codes.

When both operands are data registers, the source register contents are unpacked, the extension word is added, and the result is placed in the destination register. The high word of the destination register is unaffected.

When predecrement mode is specified, two BCD digits are extracted from a byte at the source address. After adding the adjustment, two bytes are then written to the destination address.

Example

```
PACK    -(A1),-(A2),#3
PACK    D1,D2,#3
UNPK    -(A1),-(A2),#6
UNPK    D1,D2,#6
```

4.10.19 Module Instructions (68020 only)

Syntax

CALLM #<*ddd*>,<*ea*>
RTM **R***n*

where:
<*ddd*> = the 8–bit number of bytes of arguments passed to the called module.

Call Module

An external module descriptor resides at the specified effective address location <*ea*>. This module descriptor contains control information for entry into the associated module. A module frame, containing the current module state, is created on the top of the stack. The new module state is then loaded from the external module descriptor. No condition codes are affected by this instruction. Only control alterable addressing modes are allowed.

Return from Module

A previously saved module state, from a CALLM instruction, is reloaded from the top of the stack. A register Rn is used as the module

data area pointer. If the module state includes a saved module data area pointer, register Rn is restored; else Rn is unchanged. No condition codes are affected.

Example

```
CALLM    #10,(A7)
RTM      D1
```

4.10.20 Trap on Condition Code (68020–plus and CPU32 only)

If the specified condition code is true, exception processing occurs. The vector number generated references the TRAPcc exception vector. The stacked program counter points to the next instruction. If the specified condition code is false, control passes to the next instruction in sequence. Any of the 16 condition codes may be referenced. Condition codes are not affected by this instruction. The #<*parm*> parameter in the TRAPcc form allow a 16– or 32–bit value to be embedded within the instruction for reference in exception processing.

Syntax

TRAP<*cc*>
TRAP<*cc*>[.<*size*>] #<*parm*>

where:

#<*parm*> is a 16– or 32–bit parameter used to uniquely identify a particular TRAP<*fpcc*> instruction.

<*size*> = W | L

<*cc*> = one of the condition codes described below:

T	CC (or HS)	VC	GE
F	CS (or LO)	VS	LT
HI	NE	PL	GT
LS	EQ	MI	LE

Note: T and F are not valid condition codes for conditional branch.

4.10.21 Compare and Swap with Operand (68020–plus only)

Syntax

CAS[.<*size*>] D*w*,D*y*,<*ea*>
CAS2[.<*size*>] D*w1*:D*w2*,D*y1*:D*y2*,(*Rz1*):(*Rz2*)

where:

<*size*> = B | W | L (W is default)

In the CAS instruction the specified effective address location is compared to data register Dw. If operands match, the update operand in data register Dy is written to the specified effective address location. If operands do not match, Dw is loaded with the contents at the specified effective address location.

In the CAS2 instruction, both Dw1 and Dw2 must match the values in memory pointed to by registers Rz1 and Rz2, respectively, in order that these memory values be updated by the contents of registers Dy1 and Dy2. If both operands do not match, then Dw1 and Dw2 are loaded with the contents of memory pointed to by registers Rz1 and Rz2, respectively.

Example

```
CAS     D2,D3,4(A6)
CAS2    D1:D2,D3:D4,(A3):(A4)
```

Breakpoint (68020–plus and CPU32 only)

The operation of this instruction is implementation–dependent. The processor will ask for the operation word which the breakpoint has replaced. If the operation word is furnished, the processor will execute that instruction and continue. If the operation word is not furnished, the processor will take an illegal instruction exception.

Syntax

BKPT #<*vector*>

where:

#<*vector*> specifies the breakpoint for which the processor is to request the corresponding operation word.
Value = 0 through 7.

4.10.22 Flush Entry in the Address Translation Cache (ATC) (68030 only)

The instruction PFLUSHA invalidates all ATC entries. PFLUSH invalidates a set of ATC entries whose function code (<*fc*>) bits satisfy the relation: (ATC <*fc*> & <*mask*>) = (<*fc*> & <*mask*>). Whenever <*ea*> is also specified, then the evaluated effective address must also match the specified <*ea*>.

Syntax

PFLUSHA
PFLUSH <*fc*>,#<*mask*>
PFLUSH <*fc*>,#<*mask*>,<*ea*>

where:
<*fc*> may be specified as one of the following:

immediate value	a 3–bit value in the command word
data register (Dn)	the lower 3–bit value of Dn
Source Function Code (SFC) Register	the value of the 3–bit SFC register in the CPU
Destination Function Code (DFC) Register	the value of the 3–bit DFC register in the CPU

<*mask*> indicates which bits are significant in the <*fc*> compare relation.

4.10.23 Flush Address Translation Cache (ATC) Entries (68040 and 68060 only)

The instruction PFLUSHA invalidates all entries. The PFLUSH (An) instruction invalidates the entry in each ATC which matches the logical address in An and the specified function code. The PFLUSHN (An) and PFLUSHAN instructions have a global option specified, and invalidate only non–global entries.

Syntax

PFLUSH (A*n*)
PFLUSHN (A*n*)
PFLUSHA
PFLUSHAN

4.10.24 Load an Address Translation Cache (ATC) Entry (68030 only)

The instruction PLOADR updates the used (U) bits in the address translation table as if a read access has occurred. The PLOADW

updates the used (U) and the modified (M) bits in the address translation table as if a write access has occurred.

Syntax

PLOADR *<fc>,<ea>*
PLOADW *<fc>,<ea>*

where:
<fc> may be specified as one of the following:

immediate value	a 3–bit value in the command word
data register (Dn)	the lower 3–bit value of Dn
Source Function Code (SFC) Register	the value of the 3–bit SFC register in the CPU
Destination Function Code (DFC) Register	the value of the 3–bit DFC register in the CPU

4.10.25 Move to/from MMU Registers (68030 only)

The instruction PMOVE moves the contents of the MMU register to *<ea>* or data at *<ea>* to a writable MMU register. The PMOVEFD form sets the FD bit to disable flushing the ATC when a new value is

loaded into the SRP, CRP, TT0, TT1, or TC register (but not the MMUSR).

Syntax

PMOVE.<*size*> <**MMU R***n*><*ea*>
PMOVE.<*size*> <*ea*>,<**MMU R***n*>
PMOVEFD.<*size*> <*ea*>,<**MMU R***n*>

where:
<**MMU R***n*> and <*size*> must be one of the following pairs:

CRP or SRP	D
TC, TT0, or TT1	L
MMUSR	W

4.10.26 Move to/from ACU Registers (68EC030 only)

The instruction PMOVE moves the contents of the source <ea> to the ACx register or moves the contents of the ACx register to the destination <ea>.

Syntax

PMOVE.<size> <**ACU R***n*>,<*ea*>
PMOVE.<*size*> <*ea*>,<**ACU R***n*>

where:
<**ACU R***n*> and <*size*> must be one of the following pairs:

AC0, AC1	L
ACUSR	W

4.10.27 Test a Logical Address (68030 only)

The PTESTR and PTESTW instructions get information about the logical address specified by <*fc*> and <*ea*>. If specified, **A***n* is loaded with the physical address of the last successfully fetched descriptor.

Syntax

PTESTR <*fc*>,<*ea*>,#<*level*>
PTESTR <*fc*>,<*ea*>,#<*level*>,**A***n*
PTESTW <*fc*>,<*ea*>,#<*level*>
PTESTW <*fc*>,<*ea*>,#<*level*>,**A***n*

where:
<*fc*> may be specified as one of the following:

immediate value	a 3–bit value in the command word
data register (Dn)	the lower 3–bit value of Dn
Source Function Code (SFC) Register	the value of the 3–bit SFC register in the CPU
Destination Function Code (DFC) Register	the value of the 3–bit DFC register in the CPU

<*level*> indicates the depth level to search the address translation table.

4.10.28 Test a Logical Address (68EC030 only)

The PTESTR and PTESTW instructions search the ACx registers for the address descriptor corresponding to the <ea> field and set the bit of the ACUSR according to the status of the descriptor.

Syntax

PTESTR *<fc>,<ea>*
PTESTW *<fc>,<ea>*

where:
<fc> may be specified as one of the following:

immediate value	a 3–bit value in the command word
data register (Dn)	the lower 3–bit value of Dn
Source Function Code (SFC) Register	the value of the 3–bit SFC register in the CPU
Destination Function Code (DFC) Register	the value of the 3–bit DFC register in the CPU

4.10.29 Test a Logical Address (68040 only)

The PTESTR and PTESTW instructions search the translation tables for the page descriptor corresponding to the test address in An and set the bits of the MMU status register (MMUSR) according to the status of the descriptor.

Syntax

PTESTR (A*n*)
PTESTW (A*n*)

4.10.30 Test a Logical Address (68EC040 only)

The PTESTR and PTESTW instructions search the translation tables for the page descriptor corresponding to the test address in An and set the bits of the ACU status register (ACUSR) according to the status of the descriptor.

Syntax

PTESTR (A*n*)
PTESTW (A*n*)

4.10.31 Invalidate Cache Lines (68040/68EC040 and 68060 only)

The CINVL, CINVP, and CINVA instructions invalidate selected cache lines. The CINVL invalidates the cache line (if any) matching the physical address in the specified address register. The CINVP invalidates the cache lines (if any) matching the physical memory page

in the specified address register. The CINVA instruction invalidates all cache entries.

Syntax

CINVL <*caches*>,(**A***n*)
CINVP <*caches*>,(**A***n*)
CINVA <*caches*>

where:
<*caches*> may be specified as one of the following:

DC	data cache
IC	instruction cache
BC	both caches
NC	neither cache

4.10.32 Push and invalidate Cache Lines (68040/68EC040 68060 only)

The CPUSHL, CPUSHP, and CPUSHA instructions push and then invalidate selected cache lines. When the data caches is specified, the selected data cache lines are first pushed to memory (if they contain dirty data) and then invalidated.

The CPUSHL pushes and invalidates the cache line (if any) matching the physical address in the specified address register. The CPUSHP pushes and invalidates the cache line (if any) matching the physical

memory page in the specified address register. The CPUSHA instruction pushes and invalidates all cache entries.

Syntax

CPUSHL <*caches*>,(**A***n*)
CPUSHP <*caches*>,(**A***n*)
CPUSHA <*caches*>

where:
<*caches*> may be specified as one of the following:

DC	data cache
IC	instruction cache
BC	both caches
NC	neither cache

4.10.33 Move 16 Bytes Block (68040/68EC040/68060 only)

The MOVE16 instruction moves the source line to the destination line. The lines are aligned to 16–byte boundaries. Applications for this instruction include coprocessor communications, memory initialization, and fast block copy operations.

Syntax

MOVE16 (**A***x*)+,(**A***y*)+
MOVE16 *xxx*.**L**,(**A***n*)
MOVE16 *xxx*.**L**,(**A***n*))+
MOVE16 (**A***n*),*xxx*.**L**
MOVE16 (**A***n*)+,*xxx*.**L**

4.11 Variants on Instruction Types

Certain instructions allow a "quick" and/or an "immediate" form when immediate data within a restricted size range appear as an operand. These abbreviated forms are normally chosen by the assembler, when appropriate. However, it is possible for the programmer to force such a form by appending a Q or I to the mnemonic opcode (to indicate "quick" or "immediate" respectively) on instructions for which such forms exist. If the specified quick or immediate form does not exist, or if the immediate data does not conform to the size requirements of the abbreviated form, an error is generated.

Some instructions also have "address" variant forms (which refer to address registers as destinations); these variants append an A to the instruction mnemonic (for example, ADDA, CMPA). This variant is chosen by the assembler without programmer specification, when appropriate to do so; the programmer need specify only the general instruction mnemonic. However, the programmer may "force" or specify such a variant form by appending the A. If the specified variant does not exist or is not appropriate with the given operands, an error is generated.

The CMP instruction also has a memory variant form (CMPM) in which both operands are a special class of memory references. The CMPM instruction requires postincrement addressing of both operands. The CMPM instruction will be selected by the assembler, or it may be specified by the programmer.

The variations — A, Q, I, and M — must conform to the following restrictions:

A	Must specify an address register as a destination, and cannot specify a byte size code (.B).
Q	Requires immediate operand be in a certain size range. MOVEQ also requires longword data size.
I	The size of immediate data is adjusted to match size code of operation.
M	Both operands must be postincrement addresses.

For example, the instruction:

```
ADDQ    #9,D0
```

attempts to add value 9 to D0 and causes an assembly error, because the immediate operand is not in the valid size range (1 through 8).

Although the assembler selects the appropriate opcode variations — A, Q, I, or M — when the suffix is not specified, the explicit encoding

of the suffix with the basic opcode is recommended for the following purposes:

- For documentation, to make clear in the source language the instruction form that was assembled.
- To force a format other than that which the assembler selects. For example, the assembler selects the quick (Q) form for the instruction:

```
ADD   #1,D4 Adds the value 1 to D4 via
*           an ADDQ (2-byte)
*           instruction.
```

  If the immediate (I) form is desired, the programmer must declare it explicitly, as follows:

```
ADDI  #1,D4 Adds the value 1 to D4
*           via an ADDI (4-byte)
*           instruction.
```

- To generate invariant code when using variant immediate data (separate assemblies).

4.12 68881/68882 Floating Point Coprocessors Instructions and Floating Point Instructions Supported by the 68040 and 68060

The assembler syntax that follows refers to opcodes for the 68881/68882/68040/68060.

Many 68881/68882/68040/68060 instructions can accept operands in memory. In this case, any 68020–plus addressing modes can be used to specify the address of the memory operand. All 68881/68882/68040/68060 arithmetic is done internally in 80–bit

extended precision. When a memory operand is specified, the 68881/68882/68040/68060 automatically converts it into extended precision floating point format as it is fetched from memory. In order to perform the conversion, the format of the object in memory must be specified. This is done with an extension to the mnemonic. The following extensions are available:

Extension	Format
.B	Signed byte integer
.W	Signed word integer
.L	Signed long word integer
.S	Single precision (32–bit) Real
.D	Double precision (64–bit) Real
.X	Extended precision (80–bit) Real
.P	Packed Decimal String (96–bit) Real

The floating–point instructions supported by the 68040 and 68060 are an enhanced subset of the 68881/68882 floating–point coprocessor instructions. Several instructions in the 68040 and 68060 include explicit single and double precision rounding of the result as part of their operation. For example, the FADD instruction uses the default rounding precision selected in the FPU, while FSADD and FDADD force rounding of the result to single and double precision respectively. The 68040 and 68060 directly supports portions of the floating–point instruction through hardware, and the remainder indirectly through software.

4.12.1 Floating Point Condition Codes

Floating point condition codes used in opcodes for the 68881/68882/68040/68060 are listed in the following table.

Note that the InterTools assembler supports Motorola instructions FSNE and FNE, and the aliases FSNEQ and FNEQ, respectively.

Table 2–11: 68881/68882/68040/68060–Specific Floating Point Condition Codes (fpcc)

TRAP ON UNORDERED			
fpcc		**INVERSE**	
GT	Greater than	NGT	Not greater than
GE	Greater than or equal	NGE	Not greater than or equal
LT	Less than	NLT	Not less than
GL	Greater or less than	NGL	Not less than or equal
LE	Less than or equal	NLE	Not less than or equal
GLE	Greater or less than or equal	NGLE	Not greater or less than or equal
SEQ	Equal	SNEQ	Not equal
ST	Always	SF	Never
NO TRAP ON UNORDERED			
fpcc		**INVERSE**	
OGT	Greater than	ULE	Not greater than (Unordered or less or equal)
OGE	Greater than or equal	ULT	Not greater than or equal (Unordered or less than)
OLT	Less than	UGE	Not less than (Unordered or greater or equal)
OGL	Greater or less than	UEQ	Not greater or less than (Unordered or equal)
OLE	Less than or equal	UGT	Not less than or equal (Unordered or greater than)
OR	Ordered	UN	Unordered
EQ	Equal	NEQ	Not equal (Unordered or greater or less)
T	Always	F	Never

4.12.2 Floating Point Arithmetic Operations

Syntax

F<*op*>[.<*fmt*>] <*ea*>,**FP***n*
F<*op*>**.X** **FP***m*,**FP***n*

where:
<*fmt*> = B | W | L | S | D | X | P (W is default)
<*op*> is one of the following operations:

	ADD	add
*	SADD	single precision add
*	DADD	double precision add
	DIV	divide
*	SDIV	single precision divide
*	DDIV	double precision divide
	MOD	modulo
	MUL	multiply
*	SMUL	single precision multiply
*	DMUL	double precision multiply
**	REM	remainder
**	SCALE	scale exponent
**	SGLDIV	single precision divide
**	SGLMUL	single precision multiply
	SUB	subtract
*	SSUB	single precision subtract
*	DSUB	double precision subtract
*	supported by 68040 only	

**	these instructions are software–supported on the 68040

Example

```
FADD.L      (A2),FP3
FSCALE.X    FP2,FP7
```

4.12.3 Move to Floating Point Register

This instruction loads a floating point register or floating point control register from memory or from another floating point register. The 68881/68882/68040/68060 automatically converts the source operand into extended precision floating point format as needed.

Syntax

	FMOVE[.<*fmt*>]	<*ea*>,**FP***n*
	FMOVE.X	**FP***m*,**FP***n*
	FMOVE.L	**FP***m*,<*fpcr*>
	FMOVECR.X	#<*ccc*>,**FP***n*
*	**FSMOVE**[.<*fmt*>]	<*ea*>,**FP***n*
*	**FDMOVE**[.<*fmt*>]	<*ea*>,**FP***n*

* Supported by 68040 and 68060 only

where:

<fmt> = B | W | L | S | D | X | P (W is default)

<fpcr> = CONTROL | STATUS | IADDR | FPCR | FPSR | FPIAR

<ccc> defines one of the floating point constants available in ROM

(see below).

ccc	**value**	*ccc*	**value**
00	Pi = 3.14159...	15	10**4
0B	Log10(2)	16	10**8
0C	e = 2.71828...	17	10**16
0D	Log2(e)	18	10**32
0E	Log10(e)	19	10**64
0F	0.0	1A	10**128
10	Ln(2)	1B	10**256
11	Ln(10)	1C	10**512
12	10**0 (1.0)	1D	10**1024
13	10**1	1E	10**2048
14	10**2	1F	10**4096

Example

```
FMOVE.X      FP1,FP2
FMOVECR.X    #15,FP2
FMOVE.B      (A3),FP1
FMOVE.L      (A7)+,CONTROL
```

4.12.4 Move Floating Point Register to Memory

This instruction stores a floating point register or floating point control register into memory. The 68881/68882/68040/68060 automatically converts the source operand into the required data format as needed.

Syntax

FMOVE[.<*fmt*>] **FP***n*,<*ea*>
FMOVE.L <*fpcr*>,<*ea*>
FMOVE.P **FP***n*,<*ea*>{#*kkk*}
FMOVE.P **FP***n*,<*ea*>{**D***n*}

where:
<*fmt*> = B | W | L | S | D | X (W is default)
<*fpcr*> = CONTROL | STATUS | IADDR | FPCR | FPSR | FPIAR
<*kkk*> = k–factor (see below)

If the data type format is "P" (packed decimal), then an additional parameter is required to specify the format of the resultant string. This parameter, called the "k–factor", is a 7–bit signed integer (twos complement). If less than or equal to zero it indicates the number of significant digits to the right of the decimal point (Fortran "F" format). If between 1 and 17 it indicates the number of significant digits in the mantissa (Fortran "E" format). Otherwise it is treated as 17, but the OPERR bit in the floating point status register exception byte is set. The k–factor can either be specified as an immediate or as a dynamic value taken from the low order 7 bits of a 68020–plus data register.

Example

```
FMOVE.D     FP1,(A1)
FMOVE.L     STATUS,D0
FMOVE.P     FP2,(A6){#3}
FMOVE.P     FP1,(A6,D7){D3}
```

Floating Point Move Multiple

The FMOVEM instruction moves several floating point registers or floating point control registers to or from memory.

Syntax

FMOVEM.X *<fp_reg_list>,<ea>*
FMOVEM.X **D***n*,*<ea>*
FMOVEM.X *<ea>,<fp_reg_list>*
FMOVEM.X *<ea>*,**D***n*
FMOVEM.L *<ea>,<st_reg_list>*
FMOVEM.L *<st_reg_list>,<ea>*

where:

<fp_reg_list> is a list of floating point data registers.
<st_reg_list> is a list floating point status registers.
D*n* is a data register holding the FMOVEM bit mask.

A register list consists of individual registers names separated by a slash (/) and/or ranges of floating point data registers specified by the first and last register names separated by a hyphen (–). The assembler translates this into an immediate value operand.

Example

```
        FMOVEM.X    FP0-FP3/FP7,(A7)-
*  Push FP0, FP1, FP2, FP3, and FP7 onto the
*  stack.
        MOVE.L      #$8F,D3
        FMOVEM.X    D3,(A2)
*  Last two instructions have same effect as
*  the one before.
        FMOVEM.X    (A2,D3),FP0/FP1
        MOVE.L      #$C0
        FMOVEM.X    (A2,D3),D1
*  Last two instructions have same effect
*  as the one before.
        FMOVEM.L    CONTROL/IADDR,(A7)-
        FMOVEM.L    (A7)+,CONTROL/STATUS
```

4.12.5 Floating Point Compare

Syntax

FCMP[.<*fmt*>] <*ea*>,**FP***n*
FTST[.<*fmt*>] <*ea*>
FCMP.X **FP***n*,**FP***m*
FTST.X **FP***n*

where:
<*fmt*> = B | W | L | S | D | X | P (W is default)

The FCMP instruction compares two floating point operands and sets the floating point condition codes appropriately. The FTST instruction compares the specified operand with zero. Floating point condition

codes are specified in the 68881/68882/68040/68060–Specific Floating Point Condition Codes Table 2–11.

Example

```
FTST.X      FP2
FCMP.D      12(A6),FP3
```

4.12.6 Floating Point Conditional Branch

Syntax

FB<*fpcc*>[.<*size*>] <*label*>

where:
<*fpcc*> is a floating point condition code as defined in Table 2–11.
<*size*> = W | L (L is default)

If the specified floating point condition code is met, program execution continues at <*label*>. Condition codes are not affected by this instruction.

Example

```
        FBNE.S      LABEL
        FNOP
LABEL
        FNOP
```

4.12.7 Floating Point Decrement and Branch on Condition (FDBcc)

Syntax

FDB<*fpcc*> **D***n*,<*label*>

where:
<*fpcc*> is a floating point condition code as defined in Table 2–11.

If the specified floating point condition code is met, execution continues with the next instruction. Otherwise, the low order word in the specified data register Dn is decremented by one. If the result is equal to –1, execution continues with the next instruction, else execution continues at address <*label*>. Condition codes are not affected by this instruction.

Example

```
        FDBGT       D3,LABEL1
        FNOP
LABEL1
        FNOP
```

4.12.8 Floating Point Set on Condition

The specified floating point condition code is tested. If the condition is true, the byte at the specified address location is set to TRUE (all 1's); otherwise that byte is set to FALSE (all 0's). No condition codes are

affected by this instruction. Only data–alterable addressing modes are allowed.

Syntax

FS<*fpcc*> <*ea*>

where:
<*fpcc*> is a floating point condition code as defined in Table 2–11.

Example

```
FSNE    (A3)
```

4.12.9 Floating Point Trap on Condition

Syntax

FTRAP<*fpcc*>%
FTRAP<*fpcc*>.<*size*> #<*parm*>

where:
<*size*> = W | L
<*fpcc*> is a floating point condition code as defined in Table 2–11.
<*parm*> is an immediate value.

If the selected floating point condition code is true, then the processor initiates exception processing. The vector number generated references the TRAPcc exception vector. The stacked Program Counter then points to the next instruction. If the selected floating point condition code is not true, then no operation is performed and execution

continues with the next instruction in sequence. Condition codes are not affected by this instruction.

The 16–bit or 32–bit #<*parm*> value is placed within the instruction and may be used to uniquely identify a particular FTRAP\I<*fpcc*> instruction.

Example

```
FTRAPNE
FTRAPNE.L #4
```

4.12.10 Save Coprocessor State

This instruction saves the internal state for the context switch at the specified effective address location. Condition codes are not affected by this instruction. Only predecrement or control alterable addressing modes are allowed.

Note: This is a privileged instruction.

Syntax

FSAVE <*ea*>

Example

```
FSAVE    -(A7)
```

4.12.11 Restore Coprocessor State

This instruction restores the internal state for the context switch from the specified effective address location. Condition codes are not affected by this instruction. Only postincrement or control addressing modes are allowed.

Note: This is a privileged instruction.

Syntax

FRESTORE *<ea>*

Example

```
FRESTORE (A7)+
```

4.12.12 Floating Point Functions

Syntax

F<*op*>[.<*fmt*>] <*ea*>,**FP***n*
F<*op*>**.X** **FP***m*,**FP***n*
F<*op*>**.X** **FP***n*

where:
<*fmt*> = B | W | L | S | D | X | P (W is default)
<*op*> is one of the following functions:

	ABS	absolute value
*	SABS	single precision absolute value

*	DABS	double precision absolute value
**	ACOS	arccosine
**	ASIN	arcsine
**	ATAN	arctangent
**	ATANH	hyperbolic arctangent
**	COS	cosine
**	COSH	hyperbolic cosine
**	ETOX	e**x ; @
**	ETOXM1	e**(x–1); @@
**	GETMAN	get the mantissa
**	GETEX	get the exponent
**	INT	integer part
**	INTRZ	floating point integer part, round–to–zero
**	LOGN	natural log; base e log
**	LOGNP1	natural log (x+1)
**	LOG10	common log; base 10 log
**	LOG2	binary log; base 2 log
	NEG	negate
*	SNEG	single precision negate
*	DNEG	double precision negate
**	SIN	sine
**	SINCOS	floating point simultaneous sine and cosine
**	SINH	hyperbolic sine
	SQRT	square root
*	SSQRT	single precision square root
*	DSQRT	double precision square root
**	TAN	tangent

**	TANH	hyperbolic tangent
**	TENTOX	10**x ; powers of 10
**	TWOTOX	2**x; powers of 2

*	supported by 68040 and 68060 only
**	these instructions are software–supported on the 68040
@	powers of e (Euler's constant)
@@	Euler's constant to the x–1 power

Sine/Cosine Functions

FSINCOS[.<*fmt*>] <*ea*>,**FP***c*:**FP***s*
FSINCOS.X **FP***m*,**FP***c*:**FP***s*

where:
<*fmt*> = B | W | L | S | D | X | P (W is default)
FP*s* is the floating point register holding the sine result.
FP*c* is the floating point register holding the cosine result.

Example

```
FETOXM1.W    10(A6),FP2
FABS.X       FP1,FP3
FCOSH.X      FP4
FSINCOS.B    (A6),FP1:FP2
FSINCOS.X    FP0,FP6:FP7
```

4.12.13 Floating Point NO–OP

Syntax

FNOP

This instruction is supplied for synchronization with the floating point coprocessor.

Example

```
FNOP
```

4.13 68851 Paged Memory Management Unit Coprocessor Instructions

The assembler syntax that follows refers to opcodes for the 68851.

4.13.1 Paged Memory Management Unit Condition Codes

Paged memory management unit condition codes used in opcodes for the 68851 are listed in the following table.

Table 2–12: 68851–Specific Paged Memory Management Unit Condition Codes (pcc)

SET	CLEAR	NAMED PSR REGISTER BIT
BS	BC	Bus error
LS	LC	Limit violation
SS	SC	Supervisor violation
AS	AC	Access level violation
WS	WC	Write protected
IS	IC	Invalid
GS	GC	Gate
CS	CC	Globally shared

4.13.2 Branch on PMMU Condition

If the specified PMMU condition code is met, program execution continues at address <*label*>. Processor condition codes and the PSR are n3zot affected by this instruction.

Syntax

PB<*pcc*>[.<*size*>] <*label*>

where:

<*pcc*> is a paged memory management unit condition code as defined in Table 2–12.

<*size*> = W | L (L is default)

4.13.3 Test, Decrement, and Branch on PMMU Condition

If the specified PMMU condition code is met, execution continues with the next instruction. Otherwise, the lower–order word in the specified data register Dn is decremented by one. If the result is equal to –1, execution continues with the next instruction, else execution continues at address *<label>*. Processor condition codes and the PSR are not affected by this instruction.

Syntax

PDB*<pcc>* **D***n*,*<label>*

where:

<pcc> is a paged memory management unit condition code as defined in Table 2–12.

4.13.4 Set on PMMU Condition

The specified PMMU condition code is tested. If the condition is true, the byte at the specified address location is set to TRUE (all 1's), otherwise that byte is set to False (all 0's). Processor condition codes and the PSR are not affected by this instruction.

Syntax

PS*<pcc>* *<label>*

where:

<pcc> is a paged memory management unit condition code as defined in Table 2–12.

4.13.5 Trap on PMMU Condition

If the specified PMMU condition code is true, then the processor initiates exception processing. The generated vector number references

the cpTRAPcc exception vector. The stacked program counter then points to the next instruction. If the specified PMMU condition is not true, then no operation is performed and execution continues with the next instruction in sequence. Processor condition codes and the PSR are not affected by this instruction.

Syntax

PTRAP<*pcc*>
PTRAP<*pcc*>.<*size*> #<*parm*>

where:

<*size*>	= W | L
<*pcc*>	is a paged memory management unit condition code as defined in Table 2–12.
<*parm*>	is a 16 or 32–bit immediate value used to uniquely identify a particular **PTRAP**<*pcc*> instruction.

4.13.6 Invalidate Address Translation Cache (ATC) Entries

The instruction PFLUSHA invalidates all ATC entries. PFLUSH invalidates a set of ATC entries whose function code (<*fc*>) bits satisfy the relation: (ATC <*fc*> & <*mask*>) = (<*fc*> & <*mask*>). Whenever <*ea*> is also specified, then the evaluated effective address must also match the specified <*ea*>. PFLUSHS additionally invalidates the shared globally (SG) bit of the ATC. Processor condition codes and the PSR are not affected by this instruction.

Syntax

PFLUSHA
PFLUSH <*fc*>,#<*mask*>
PFLUSH <*fc*>,#<*mask*>,<*ea*>

PFLUSHS *<fc>,#<mask>*
PFLUSHS *<fc>,#<mask>,<ea>*

where:
<fc> may be specified as one of the following:

immediate value	a 4–bit value in the command word
data register (Dn)	the lower 4–bit value of Dn
Source Function Code (SFC) Register	the value of the 3–bit SFC register in the CPU
Destination Function Code (DFC) Register	the value of the 3–bit DFC register in the CPU

<mask> indicates which bits are significant in the *<fc>* compare relation.

4.13.7 Invalidate Address Translation Cache (ATC) and Root Pointer Table (RPT) Entries

The double longword at *<ea>* is used as the previous value of the CRP register. Matching RPT entries in this saved CRP value are flushed along with all ATC entries loaded with the saved CRP value (except for globally shared entries). If a match does not exist between the value at *<ea>* and the corresponding RPT, then no action results. Processor condition codes and the PSR are not affected by this instruction.

Syntax

PFLUSHR *<ea>*

4.13.8 Load an Address Translation Cache (ATC) Entry

The instruction PLOADR updates the used (U) bits in the address translation table if a read access has occurred. The PLOADW updates the used (U) and the modified (M) bits in the address translation table if a write access has occurred. Processor condition codes and the PSR are not affected by this instruction.

Syntax

PLOADR *<fc>,<ea>*
PLOADW *<fc>,<ea>*

where:
<fc> may be specified as one of the following:

immediate value	a 4–bit value in the command word
data register (Dn)	the lower 4–bit value of Dn
Source Function Code (SFC) Register	the value of the 3–bit SFC register in the CPU
Destination Function Code (DFC) Register	the value of the 3–bit DFC register in the CPU

4.13.9 Move PMMU Register

The instruction PMOVE moves the contents of the PMMU register to *<ea>* or data at *<ea>* to a writable PMMU register. Processor

condition codes and the PSR are not affected by this instruction except where the PSR has been written to.

Syntax

PMOVE.<*size*> **<PMMU R***n*>,<*ea*>
PMOVE.<*size*> <*ea*>,**<PMMU R***n*>

where:
<PMMU R*n*> and <*size*> must be one of the following pairs:

CRP, SRP, or DRP	D
TC	L
BACx, BADx, AC, PSR, or PCSR	W
CAL, VAL, or SCC	B

4.13.10 PMMU Restore Function

The PRESTORE instruction restores the nonuser visible state of the MC68851 as well as PSR, CRP, SRP, CAL, VAL, and SCC registers of the user programming model. Additionally, BACx and BADx registers are restored if any breakpoints are enabled. The PSAVE instruction is the inverse of the PRESTORE instruction. Processor condition codes are not affected by this instruction. The PSR is set according to the data restored.

Note: This is a privileged instruction.

Syntax

PRESTORE <*ea*>

4.13.11 PMMU Save Function

The PSAVE instruction saves the internal state and certain programmer registers in a state frame located at the <*ea*>. PSR, CRP, SRP, CAL, VAL, SCC, and all BACx and BADx (if any breakpoints are enabled) are the PMMU registers saved. PSAVE is the inverse of PRESTORE. Processor condition codes and the PSR are not affected by this instruction.

Note: This is a privileged instruction.

Syntax

PSAVE <*ea*>

4.13.12 Get Information about Logical Address

The PTESTR and PTESTW instructions get information about the logical address specified by <*fc*> and <*ea*> if the enable (E) bit of the TC register is set. PTESTR updates the access level violation (A) bit on a read while PTESTW updates the access level violation (A) bit on a write. If specified, **A**n is loaded with the physical address of the last successfully fetched descriptor. The PSR is affected by this instruction.

Syntax

PTESTR <*fc*>,<*ea*>,#<*level*>
PTESTR <*fc*>,<*ea*>,#<*level*>,**A***n*
PTESTW <*fc*>,<*ea*>,#<*level*>
PTESTW <*fc*>,<*ea*>,#<*level*>,**A***n*

where:
<*fc*> may be specified as one of the following:

immediate value	a 4–bit value in the command word
data register (Dn)	the lower 4–bit value of Dn
Source Function Code (SFC) Register	the value of the 3–bit SFC register in the CPU
Destination Function Code (DFC) Register	the value of the 3–bit DFC register in the CPU

<level> indicates the depth level to search the address translation table.

4.13.13 Validate a Pointer

The PVALID instruction compares the upper bits of the **VAL** or **A***n* register with the upper bits of the *<ea>*. If the number of upper bits (predefined in the ALC field of the AC register) of the **VAL** or **A***n* is greater than the destination, then an MMU access level exception occurs. Processor condition codes and the PSR are not affected by this instruction.

Syntax

PVALID VAL,*<ea>*
PVALID A*n*,*<ea>*

CHAPTER 5

Assembler Directives

All assembler directives (pseudo–ops), with the exception of DC and DCB, are instructions to the assembler rather than instructions to be translated into object code. This section contains descriptions and examples of the basic forms of the most frequently used assembler directives. See also the *Macro and Conditional Assembly* and *Structured Control Statements* sections about directives. The most commonly used directives supported by the assembler are Assembly Control, Symbol Definition, Data Definition/Storage Allocation, Listing Control and Output Options, External Symbol Control, and Internal Assembly Control.

5.1 Assembly Control

The InterTools Assembler contains the following assembly control directives:

COMMON	Named Common
END	Program End
INCLUDE	Include Secondary File
OFFSET	Define Offsets
ORG	Absolute Origin
RESERVE	Reserve room in section
RESUME	Resume section
RORG	Relocatable Org
SECTION	Relocatable Program Section
_BRINGIN	Declare external symbol
_DEBSYM	Put out debugging information
_DGROUP	Define data group

5.1.1 COMMON – Enter Named Common Section

Syntax

[*label*]**COMMON**[**.S**]<*name*> [,[**ABSOLUTE**[:[<*location*>]],"<*class*>"]

The COMMON directive performs the same functions as the SECTION directive which is described later in this chapter, except that the generated section is marked as "common" in the object module. When the linker encounters the same common section name in two object modules, it combines them by overlay; the length of the resulting section is the maximum of the lengths of the input sections.

In contrast, when the linker encounters the same non–common section in two object modules, in combines them by *concatenation*; the length of the resulting section is then the sum of the lengths of the input sections.

Note: This directive can only be used if the "select TASKING extensions" switch or option is in effect.

If the ABSOLUTE specification is selected, the segment defined by the COMMON statement will be located at the indicated absolute address (or zero, if <*location*> is omitted). If the same absolute common section is defined in more than one assembly, then the same <*location*> must be specified at every definition.

If present, the .S indicates the section will be placed completely within low address memory, i.e., between addresses 0 and hex 7FFF or between hex FFFF8000 and FFFFFFFF. This allows the assembler to generate more efficient addressing to items within the section. In particular, it can implement direct addressing through the "absolute short" addressing mode. Unlike the ABSOLUTE attribute described above, this information is *not* passed on in the object module; the user must take responsibility for placing the section in low memory with the InterTools locator.

<*class*> is an arbitrary string that will be associated with the section in the object module. The locator can locate all the sections with a given class name with a single command, so a consistent use of class names may make it easier to locate your program in memory. Please see the *Linking Locator* chapter in the *InterTools User's Manual* for more details.

Example

```
clab1    COMMON   sect1,,"cclass"
clab2    COMMON   sect2,ABSOLUTE:3000
clab3    COMMON   sect3,ABSOLUTE:100,"myclass"
```

5.1.2 END – Program End

Syntax

END [<*start address*>]

The END directive indicates to the assembler that the source is finished. Subsequent source statements are ignored. The END directive encountered at the end of the first pass through the source program causes the assembler to start the second pass. The start address should be specified unless it is external to the module. If no

start address is specified, it is still possible to include a comment field, provided the comment field is set off by the exclamation point (!).

This syntax indicates to the assembler that the operand field is null and that a comment field follows.

The END statement is optional.

Example

```
in file1.68k:
        SECTION FOO
        NOP
        END       !  end of program, no
*                    starting point defined

in file2.68k:
        SECTION BAR
        __BEGIN
        NOP
        END       __BEGIN
* end of program, program starts at __begin
```

5.1.3 INCLUDE – Include Secondary File

Syntax

INCLUDE *<file>* Include file from system directory.
INCLUDE *file* Include file from user directory.

The INCLUDE directive is inserted in the source program at any point where a secondary file is to be included in the source input stream. The first column on the line with the INCLUDE directive **must** be white space.

The search algorithm for finding the *<file>* depends on whether the file is to be found in the system directory list (specified in a command

line switch) or the standard include directory list (specified in a command line switch).

Directories Searched:

<*file*> System directories only.
file User and System directories.
* Current directory.

Example

```
*  The first column in the include line must be
*  "white space"
        INCLUDE     local_file.inc
*
*  Now include a system level include file
        INCLUDE     <system_file.inc>
```

5.1.4 OFFSET – Define Offsets

Syntax

OFFSET <*expression*>

The OFFSET directive is used to define a table of offsets via the Define Storage (DS) directive without actually declaring storage for the table, in effect creating a dummy section. Symbols defined in an OFFSET table are kept internally, but will not appear in the output object module. No code producing instructions or directives may appear. SET, EQU, REG, XDEF directives are allowed.

<*expression*> is the value at which the offset table is to begin. The expression must be absolute and may not contain forward, undefined, or external references.

OFFSET must be terminated by an ORG or SECTION directive before further code producing instructions are generated. If not, the assembler produces an error message.

Example

```
          OFFSET   $7FFF
OFF1      DS.W     1          *  OFF1 is defined to
                              *  be 8000 hex
OFF2      DS.W     1          *  OFF2 is defined to
                              *  be 8002 hex

          OFFSET   $0
OFF3      DS.W     1          *  OFF3 is defined to
                              *  be 0
OFF4      DS.W     1          *  OFF4 is defined to
                              *  be 2

          SECTION  MORECODE
          LEA      OFF1,A1   *  LEA 8000,A1
          LEA      OFF2,A1   *  LEA 8002,A1
          LEA      OFF3,A1   *  LEA 0,A1
          NOP
          END
```

5.1.5 ORG – Absolute Origin

Syntax

ORG[.<*qualifier*>]<*expression*>[<*comment*>]

where:
<*qualifier*> is S | L

The ORG directive changes the program counter to the value specified by the expression in its operand field. Subsequent statements are assigned absolute memory locations starting with the new program

counter value. <*expression*> must be absolute and may not contain any forward, undefined, or external references.

ORG.S is interpreted as both ORG and OPT FRS (Forward Reference Short Option). ORG.L is interpreted as both ORG and OPT FRL (Forward Reference Long Option). Regardless of the forward reference option, references to previously defined absolute symbols will always generate the appropriate short or long addressing form, based upon the size of a symbol's absolute address.

5.1.6 RESERVE – Reserve storage

Syntax

[*label*] **RESERVE** <*name*>, <*length*> [,"<*class*>"]

The RESERVE directive is similar to the OFFSET directive. It defines a segment and gives it the specified length, but leaves the user in the previous segment. Subsequent RESERVE directives with the same <*name*> add to the segment length.

Note: This directive can only be used if the "select TASKING extensions" option is in effect. See the *68000 Family Assembler* chapter in the *InterTools User's Manual* for more information on the TASKING extensions.

<*class*> is an arbitrary string that will be associated with the section in the object module. The InterTools locator can locate all the sections with a given class name with a single command, so a consistent use of class names may make it easier to locate your program in memory. See

the *Linking Locator* chapter in the *InterTools User's Manual* for more details.

Example

```
        SECTION     DSCT
TOP     RESERVE     RSECT,2
        DC.L        TOP
HERE    RESERVE     RSECT,3
        DC.L        HERE
THERE   RESERVE     RSECT,1
        DC.L        THERE
```

This creates a segment, RSECT, of length six and defines labels at locations 0, 2, and 5 within it. The addresses of the labels are stored in DSCT.

5.1.7 RESUME – Resume defined section

Syntax

RESUME *<name>*

The RESUME directive resumes the named section. If no argument is given, the PSCT section is resumed.

Note: This directive can only be used if the "select TASKING extensions" option is in effect. See the *68000 Family Assembler* chapter in the *InterTools User's Manual* for more information on the TASKING extensions.

Example

```
SECTION     TOP
NOP
SECTION     BOTTOM
NOP
RESUME      TOP
NOP
```

5.1.8 RORG – Relocatable ORG

Syntax

RORG <*expression*> [<*comment*>]

The RORG directive has an effect similar to ORG, but it is intended for use in relocatable sections. The location counter is set to the value of the expression but remains in the current section. This is different from ORG, which switches to absolute assembly.

Note: This directive can only be used if the "select TASKING extensions" option is in effect. See the *68000 Family Assembler* chapter in the *InterTools User's Manual* for more information on the TASKING extensions.

Example

```
DIGINX   EQU.B    0
ISDIGIT  DCB.B    256,0 Define table of 256 zeros.
*  Reset location counter back to ISDIGIT
         RORG     ISDIGIT
*  Advance to index of first ASCII digit
         RORG     *+DIGINX
*  Redefine table contents to be 1's at index of
*  ASCII digits
         DCB.B    10,1
         RORG     ISDIGIT+256     Reset location
*                                 counter past
*                                 ISDIGIT
         ...
         LEA      ISDIGIT,A1      Load address of
*                                 table
         MOVEQ.L  #0,D0
         MOVE.B   NEXTCHAR,D0     Pick up a
*                                 character
         BNZ      (A1,D0.W),L1    Goto L1 if it's a
*                                 digit
         BRA   L2                 Goto L2 otherwise
```

5.1.9 SECTION – Relocatable Program Section

Syntax

SECTION[**.S**] <*name*>[,[**ABSOLUTE** [:<*location*>]][,"<*class*>"]]
SECTION[**.S**] <*number*>

This directive causes the program counter to be restored to the address following the last location allocated in the indicated section, or to zero if used for the first time with this <*name*>.

Note: <location> and "class" can only be supplied if the "select TASKING extensions" option is in effect. See the *68000 Family Assembler* chapter in the *InterTools User's Manual* for more information on the TASKING extensions.

If the ABSOLUTE specification is selected, the segment defined by the SECTION statement will be located at the indicated absolute address (or zero, if <*location*> is omitted).

Note: Absolute sections are "uncombinable", so it will cause an error if the same section is defined in another assembly or compilation. See the *Linking Locator* chapter in the *InterTools User's Manual* for a description of segment combinability.

The .S indicates the section will be placed completely within low address memory, i.e., between addresses 0 and 32767. This allows the assembler to generate more efficient addressing to items within the section. In particular, it can implement direct addressing through the absolute short addressing mode. Unlike the absolute attribute described above, this information is *not* passed on in the object module; the user must take responsibility for placing the section in low memory with the InterTools locator.

<*number*> causes a segment named $$seg<*number*> to be created.

<*class*> is an arbitrary string that will be associated with the section in the object module. The InterTools locator can locate all the sections with a given class name with a single command, so a consistent use of class names may make it easier to locate your program in memory. Please see the *Linking Locator* chapter in the *InterTools User's Manual* for more details.

Example

```
SECTION  lost
SECTION  abs1,ABSOLUTE,"abclass"
SECTION  abs2,ABSOLUTE
SECTION  found2,,"foundclass"
SECTION  2
```

5.2 Symbol Definition

Symbol definition directives EQU, FEQU, REG, and SET provide the only method by which a symbol appearing in the label field may be assigned a 'value' other than that corresponding to the current location counter. The following Symbol Definition Directives are described in this section:

EQU	Equate Symbol Value
FEQU	Equate Floating Point Symbol Value
REG	Define Register List
SET	Set Symbol Value

5.2.1 EQU – Equate Symbol Value

Syntax

<*label*> **EQU** <*expression*> [<*comment*>]

The EQU directive assigns the value of the expression in the operand field to the symbol in the label field. The label and expression follow the rules given in the *Source Program Coding* chapter. The label and operand fields are both required, and the label cannot be defined anywhere else in the program.

Any valid expression is allowed in the operand field of an EQU, including forward and complex.

Example

```
STRT    EQU         *  This is the start location
```

5.2.2 FEQU – Equate Floating Point Symbol Value

Syntax

<*label*> **FEQU**[.<*size*>] <*value*> [<*comments*>]

where:
<*size*> = S | D | X | P (S is default)

FEQU directive assigns the floating point value in the operand field to the symbol in the label field. The label and value follow the rules given in the *Source Program Coding* chapter. The operand fields are both required, and the label cannot be defined anywhere else in the program. Note that <*value*> is stored as a string and only converted to

its binary format when it is used in instructions. <*value*> may be a floating point decimal string or a floating point hexadecimal value as defined in the *Source Program Coding* chapter. A warning is generated whenever the number of bits required to represent the specified precision is exceeded.

Example

```
OP1     FEQU.X      2.3444
OP2     FEQU.S      :23444
```

5.2.3 REG – Define Register List

Syntax

<*label*> **REG** <*reg_list*> [<*comment*>]

The REG directive assigns a value to <*label*> that can be translated into the register list mask format used in the MOVEM instruction. The label cannot be redefined as a Class 2 symbol anywhere else in the program.

<*reg_list*> is of the form R1[–R2][/R3[–R4]]...

Example

```
SAVE    REG         A1-A5/D0/D2-D4/D7
*  Following two statements are then equivalent
        MOVEM.L     SAVE,-(A7)
        MOVEM.L     A1-A5/D0/D2-D4/D7,-(A7)
```

5.2.4 SET – Set Symbol Value

Syntax

<label> **SET** *<expression>* [*<comments>*]

SET directive assigns the value of the expression in the operand field to the symbol in the label field. Thus, the SET directive is similar to the EQU directive. However, the SET directive allows the symbol in the label field to be redefined by other SET directives in the program. The label and operand fields are both required.

As with EQU, any valid expression is allowed in the operand field of a SET, including forward and complex.

Example

```
THIRTY   SET          LAB_AT_30
```

5.3 Data Definition/Storage Allocation

The directives in this section provide the only means by which object code may begin or end on odd byte boundaries. All instructions and all word or longword size data must begin and end on even byte boundaries. Odd byte alignment is allowed only for the DC.B, DS.B, and DCB.B directives. All other operations which generate object code are preceded by a zero fill byte if word boundary alignment is required.

The following directives are described in this section:

COMLINE	Command Line (unimplemented)
DC	Define Constant
DCB	Define Constant Block
DS	Define Storage

5.3.1 COMLINE – Unimplemented

The COMLINE directive is not implemented. It is read and ignored. In the Motorola assembler it allows the user to specify the command line.

5.3.2 DC – Define Constant

Syntax

[<*label*>] **DC**[.<*fmt*>] <*operand*>[,<*operand*>...]

where:
<*fmt*> = B | W | L | S | D | X | P (W is default)
<*operand*> = link–time constant expression

The DC directive defines a constant in memory. The DC directive may have one or more operands, which are separated by commas. The operand field may contain the actual value (decimal, hexadecimal, or ASCII). Alternatively, the operand may be a symbol or expression which can be evaluated either by the assembler or the linker. The constant is aligned on a word boundary if word (.W), longword (.L), single precision (.S), double precision (.D), extended precision floating point (.X) or packed BCD (.P) is specified. Alignment is on a byte

boundary if byte (.B) is specified. The type of the operand must be floating point if and only if the format is S, D, X, or P.

The following rules apply to size specifications on DC directives with ASCII strings as operands:

- DC.B
  One byte is allocated per ASCII character.
- DC.W
  The string begins on a word boundary. If the string address contains an odd number of characters, a zero fill byte follows the last character.
- DC.L
  The string begins on a word boundary. If the string length is not a multiple of four bytes, the last longword is zero filled.

Examples of ASCII Strings

Directive	Result
DC.B 'ABCDEFGHI'	Memory has nine contiguous bytes with the ASCII characters A through I.
DC.B 'E'	Memory has characters "EJ" ($454A) in DC.B 'J' contiguous bytes.
DC.B 'E'	Memory has $45004500 in contiguous bytes, DC.W 'E' the first zero byte being an odd byte fill as outlined above.
DC 'X'	Memory has $5800 in contiguous bytes.
DC.L '12345'	Memory has $3132333435000000 in contiguous bytes.

Examples of Numeric Constants

Directive	Result

DC.B 10,5,7	Memory has three contiguous bytes with the decimal values 10, 5, and 7 in their respective bytes.
DC.W 10,5,7	Each operand is contained in a word. The value 10 is contained in the first word, right justified. The value 5 is in the second word, and the value 7 is in the third word.
DC.L 10,5,7	Each operand is contained in a longword. The value 10 is contained in the first longword (4bytes) right justified. The value 5 is in the second longword, and the value 7 is in the third longword.
DC LABEL+1	The generated value is the address of LABEL plus 1 in a word size operand.
DC $FF,$10,$AE	Rules for hexadecimal are the same as decimal.
DC.S 3.1415	A single precision floating point value is created. (68881/68882/68040/68060 only)
DC.D 2.54	A double precision floating point value is created. (68881/68882/68040/68060 only)
DC.X 6.0224E23	An extended precision floating point value is created. (68881/68882/68040/68060 only)
DC.X :ABCD10	An extended precision floating point hex value is created. (68881/68882/6804068060 only)
DC.P 3.00E9	A packed BCD value is created.

Note: For DC.X, "E" can only be a hex digit, not an exponent.

If the resulting value in an operand expression exceeds the size of the operand, an error is generated. For example,

DC.B $FFF	This causes an error because $FFF cannot be represented in 8 bits.
DC $FFF6F	This causes an error because $FFF6F cannot be represented in 16 bits.

5.3.3 DCB – Define Constant Block

Syntax

[<*label*>] **DCB**[.<*size*>] <*length*>,<*value*>

where:

<*size*> = B | W | L | S | D | X | P (W is default)
<*value*> = integer, character, or floating point value

DCB directive causes the assembler to allocate a block of bytes, words, or long words, quad words (.D), or hex words (.X or .P) depending upon the <*size*> specified. If <*size*> is omitted, word (.W) is the default size. The block length is specified by the absolute expression <*length*>, which may not contain undefined, forward, or external references. The initial value of each storage unit allocated will be the sign–extended expression <*value*>. <*value*> may be relocatable and may contain forward references. <*length*> must be greater than zero.

Example

```
CLEAR    DCB   80,0  Clears one line to spaces
```

5.3.4 DS – Define Storage

Syntax

[<*label*>] **DS**[.<*fmt*>] <*objects*>

where:

<*fmt*> = B | W | L | S | D | X | P (W is default)
<*objects*> = The number of objects to be reserved (a pass 1 constant).

The DS directive is used to reserve memory locations. The contents of the memory reserved are not initialized in any way. The <*fmt*> values of S, D, X, and P are only used for floating point values, and so only apply when assembling for the 68881/68882/68040/68060.

Example

```
        DS.B    10      Reserve 10 bytes
PT1     DS      $10     Reserve 16 words
PT2     DS.L    100     Reserve 100 long words
        DS.D    10      Reserve 10 8-byte
*                       double words
```

The label will reference the lowest address of the defined storage area. The storage area is aligned to a word boundary unless <*fmt*> is "B".

```
        DS.B    1       Reserve one byte
        DS      0       Set location counter
*                       to even boundary
```

The operand must be absolute and may not contain forward, undefined, or external references.

5.4 Listing Control and Output Options

The following Listing Control and Output Options are described in this section:

FAIL	Programmer Generated Error
FORMAT/NOFORMAT	Format Options
LIST/NOLIST	List Options
LLEN	Line Length
NOOBJ	No Object
OPT	General Option Selection
PAGE/NOPAGE	Pagination Options
SPC	Space Between Source Lines
STTL	Subtitle
TTL	Title

5.4.1 FAIL – Programmer Generated Error

Syntax

FAIL <*message*>

The FAIL directive causes a warning message to be printed by the assembler. The FAIL directive is normally used in conjunction with conditional assembly directives for exceptional condition checking. The assembly proceeds normally after the warning has been printed. The argument is printed as the warning message.

5.4.2 FORMAT/NOFORMAT – Unimplemented

FORMAT and NOFORMAT are not implemented. They are read and ignored. In the Motorola assembler it allows the user to control the formatting of the assembler's listing file.

5.4.3 LIST/NOLIST – Control Listing Generation

Syntax

LIST
NOLIST
NOL

Print or do not print the assembly listing on the output device. The LIST option is selected by default. The source text following the LIST directive is printed until an END or NOLIST directive is encountered.

5.4.4 LLEN – Unimplemented

The LLEN directive is not implemented. It is read and ignored. In the Motorola assembler it allows the user to specify the length of a listing line.

5.4.5 NOOBJ – Unimplemented

The NOOBJ directive is not implemented. It is read and ignored. In the Motorola assembler it allows the user to request that no object module be produced.

5.4.6 OPT – Assembler Options

Syntax

OPT <*option*>[,<*option*>...] [<*comment*>]

- Follows the command format. The available options are:

  A Absolute address. All non–indexed operands which reference either labels or the current assembler location counter (*) is resolved as absolute.

NOA	Disable A (default).
BRB **BRW**	Generates default branch size of 16 bits.
BRL	Forward branch long (default). Forward references in relative branch instructions (Bcc, BRA, BSR) will assume the longer form (16–bit displacement, yielding a 4–byte instruction). A 32–bit displacement is assumed unless the directive OPT OLD is in effect (68020–plus/CPU32 only).
BRS	Forward branch short. As with BRL, but using the shorter or form (8–bit displacement, yielding a 2–byte instruction).
CRE	Print cross–reference table at end of source listing. This option must precede first symbol in source program. If this option is not in effect, only the symbol table is printed.
EXT	Not implemented.
NOEXT	Not implemented.
FRL	Forward reference long (default). Forward references in the absolute format assumes absolute long mode (32–bit).
FRS	Forward reference short. Forward references in the absolute format assumes absolute short mode (16–bit).
L	Turn on source listing.
NOL	Turn off source listing.
MC	Print macro calls (default).

NOMC	Opposite of MC.
MD	Print macro definitions (default).
NOMD	Opposite of MD.
MEX	Print macro expansions.
NOMEX	Opposite of MEX (default).
MEXG	List only those macro expansions that generate code.
NOMEXG	Turn off MEXG flag.
OLD	Interpret the branch size code .L as being a 16–bit branch. Also interpret future uses of OPT BRL as referring to forward 16–bit branches.
NOOLD	Change back to new branch size meanings for size .L (68020–plus/CPU32 only).
P	Turn on listing paging.
NOP	Turn off listing paging.
PCO	PC relative addressing within ORG. Employ relative addressing, when possible, on backward references occurring in an ORG section.
NOPCO	Disable PCO (default).

PCS	Force PC relative addressing within SECTION. Forces PC relative addressing (whenever such an addressing mode is legal) in an instruction which occurs within a relocatable SECTION and references an operand in a relocatable SECTION (need not be the same SECTION as the instruction). Failure to resolve such a reference into a 16–bit displacement from the PC results in an error. This option may be used to force position–independent code (refer to the *Position–independent Code* chapter); however, this option does not force PC relative addressing of absolute operands (defined in ORG section) or unknown forward references.
NOPCS	Disable PCS (default).
P=<type>	Not implemented.
S	Turn on symbol table listing.
NOS	Turn off symbol table listing.
PSA	List expanded instructions from structured assembler constructs.
NOPSA	Turn off expanded instruction listing.
TRM	Trim comments from listing.
NOTRM	Do not trim comments from listing.
U	Turn on listing of unassembled lines in conditional assembly.

NOU Turn off listing of unassembled lines in conditional assembly.

- The following options are **not** implemented:

ASM
NOASM
CEX
NOCEX
CL
NOCL
D To generate debug output, use the command–line debugging option. See the *InterTools Tutorial* chapter of the *InterTools Getting Started Manual* for more information.

EQU
NOEQU
FMT
NOFMT
G
NOG
LLE
O
NOO
REL

5.4.7 PAGE/NOPAGE – Control Pagination

Syntax

PAGE [*size*]
NOPAGE

Advance the paper to the top of the next page. The PAGE directive does not appear on the program listing. No label is used, and no

machine code results. The optional size argument is used as the number of lines per page. A negative value for <*size*> turns off paging.

NOPAGE turns off pagination. Output lines are printed continuously with no page headings or top and bottom margins.

5.4.8 SPC – Space Between Source Lines

Syntax

SPC [*n*]

Output *n* blank lines on the assembly listing. This has the same effect as putting *n* blank lines in the assembly source. The default value for *n* is 1.

5.4.9 STTL – Set Subtitle

Syntax

STTL <*subtitle string*>

Print the <*subtitle*> string on the second line of each page. A subtitle consists of up to 60 characters. The same subtitle will appear on all successive pages until another STTL directive is encountered. In order to print a subtitle on the first listing page, the STTL directive must precede the first source line which will appear on the listing.

5.4.10 TTL – Set Title

Syntax

TTL *<title string>*

Print the *<title>* string at the top of each page. A title consists of up to 60 characters. The same title will appear at the top of all successive pages until another TTL directive is encountered. In order to print a title on the first listing page, the TTL directive must precede the first source line which will appear on the listing.

5.5 External Symbol Controls

The following External Symbol Controls are described in this section:

IDNT	Relocatable Identification Record
XDEF	External Symbol Definition
XREF	External Symbol Reference

5.5.1 IDNT – Relocatable Identification Record

Syntax

<name> **IDNT** *<version_string>*

The assembler takes the provided information and puts it in the object module as a .ID statement. This statement is ignored by subsequent processors, but is passed on for informational purposes only. *<version>* must be supplied as a quoted string.

Note: <name> is NOT considered a label, and may not be used elsewhere in the assembly.

5.5.2 XDEF – External Symbol Definition

Syntax

XDEF <*symbol*>[,<*symbol*>...] [<*comment*>]

The XDEF directive specifies symbols defined in the current module that are to be globally visible, and can therefore be referenced by other modules.

Example

```
        XDEF        var1,var2,var2
*  These names may now be referenced in other
*  modules
```

5.5.3 XREF – External Symbol Reference

Syntax

XREF[**.S**] [[<*section*>:]<*symbol*>[,<*symbol*>]...]

This directive specifies symbols referenced in the current module but defined in other modules. Each symbol is associated with the specified <*section*> number which it follows. Symbols may occur in any

section, including an absolute ORG section, if no *<section>* designation is specified.

".S" indicates the XREF symbols should be directly addressed through absolute short mode. Remember, however, that the location of the symbols in low memory is the responsibility of the user.

Example

```
XREF    Simple_var
XREF    AA,2:E2,3:E3,B3,C3
```

The symbol AA can be in any section; E2 is in section 2; and E3, B3, and C3 are in section 3.

5.6 Internal Assembly Controls

The directives that are described in this section are put out by our 68000 family C compiler. Compiling with the "–ia" switch, the compiler will produce real assembly language output. These directives are needed to provide the connection between the compiler produced assembly and the compiler libraries and to pass symbolic debug information through the assembler:

_BRINGIN
_DEBSYM
_DGROUP

5.6.1 _BRINGIN Declare external symbol

Syntax

_BRINGIN *<symbol>*

This directive tells the assembler to emit an external reference for the named symbol into the object module. This causes the link editor to bring the object module that defines this symbol into the link.

5.6.2 _DEBSYM Put out debugging information

Syntax

_DEBSYM *<string>*[,*<operand>*]

This directive causes the assembler to generate a line of symbolic debugging information. Lines that describe data symbol positions are written to a temporary file which is later combined into the object module by the compiler utility. Line that describe C source line positions are written directly into the object module.

5.6.3 _DGROUP Define data group

Syntax

_DGROUP *<symbol>*

The compiler addresses global data via the A5 register. The global data is divided into two segments, idata for initialized data and udata for unitialized data. In order to address two different segments off one register, a virtual segment or "group" called "data" is used. The A5 register points at data, and the link editor ensures that idata and udata are located in one 64k byte area.

This directive causes the assembler to emit a group definition into the object module. The name symbol (always "data") is the group name. The statement puts the udata and/or idata segments into the group data if they are present.

CHAPTER 6

Macro Operations and Conditional Assembly

This chapter describes the macro and the conditional assembly capabilities of the assembler. These features can be used in any program.

6.1 Macro Operations

Programming applications frequently involve the coding of a repeated pattern of instructions that, within themselves, contain variable entries at each iteration of the pattern, or basic coding patterns subject to conditional assembly at each occurrence. In either case, macros provide a shorthand notation for handling these patterns. Having determined the iterated pattern, the programmer can, within the macro, designate fields of any statement as variable. Thereafter, by invoking a macro, the programmer can use the entire pattern as many times as needed, substituting different parameters for the designated variable portions of the statements.

Macro usage can be divided into two basic parts: definition and expansion.

When the pattern is defined, it is given a name. This name becomes the mnemonic by which the macro is subsequently invoked (called). The name of a macro definition should not be the same as an existing instruction mnemonic an assembler directive, or a previously defined macro.

Expansion occurs when the previously defined macro is called (invoked). The macro call causes source statements to be generated. The generated statements may contain substitutable arguments. The statements that may be generated by a macro call are relatively unrestricted as to type. They can be any processor instruction, almost any assembler directive, or any previously defined macro. Source statements generated by a macro call are subject to the same conditions and restrictions as programmer–generated statements.

The invocation of a macro requires that the macro name appear in the operation field of a source statement. Most arguments are placed in the operand field. Appropriate arguments selected according to the macro definition cause the assembler to produce in–line coding variations of the macro definition.

The effect of a macro call is the same as an open subroutine in that it produces in–line code to perform a predefined function. The in–line code is inserted in the normal flow of the program so that the generated instructions are executed in–line with the rest of the program each time the macro is called.

6.1.1 Macro Definition

The definition of a macro consists of three parts:

1. The header: <*label*> MACRO

   The <*label*> of the MACRO statement is the "name" by which the macro is later invoked. This name must be a unique class 1 symbol. A macro name may not have a period (.) as any character other than the first.

2. The body

   The body of a macro is a sequence of standard source statements. Macro parameters are defined by the appearance of argument designators within these source statements. Legal macro–generated statements include the set of Motorola 68000 family assembly language instructions, assembler directives, structured syntax statements, and calls to other, previously defined macros. However, macro definitions may not be nested.

3. The terminator: ENDM

6.1.2 Macro Invocation

The form of a macro call is:

[<*label*>] <*name*>[.<*qualifier*>] [<*parameters*>]

Although a macro may be referenced by another macro prior to its definition in the source module, the macro must be defined before its first in–line expansion. The name of the called macro appears in the operation field of the source statement; parameters may appear as

qualifiers to the macro name and/or in the operand field of the source statement, separated by commas.

The macro call produces in–line code at the location of the invocation, according to the macro definition and the parameters specified in the macro call. The source statements so generated are then assembled, subject to the same conditions and restrictions affecting any source statement. Nested macro calls are also expanded at this time.

6.1.3 Macro Parameter Definition and Use

Up to 36 different, substitutable arguments may appear in the source statements which constitute the body of a macro. These arguments are replaced by the corresponding parameters in a subsequent call to that macro.

Arguments are designated by a backslash character (\), followed by a digit (0 through 9) or an upper case letter (A through Z). Argument designator \0 refers to the qualifier appended to the macro name; parameters in the operand field of the macro call refer to argument designations \1 through \9 and \A through \Z, in that order.

Argument substitution at the time of a macro call is handled as a literal (string) substitution. The string corresponding to a given parameter is substituted literally whenever that argument designator occurs in a source statement as the macro is expanded. Each statement generated in this expansion is assembled in–line. (Note that, if a qualifier is present, argument 0 begins with the first character following the period which separates the qualifier from the macro name).

It is possible to specify a null argument in a macro call by an empty string (not a blank); except for 0, it must still be separated from other parameters by a comma. In the case of a null argument referenced as a size code, the default size code (W) is implied; when a null argument itself is passed as an argument in a nested macro call, a null argument is passed. All parameters have a default value of null at the time of macro call.

If an argument has multiple parts or contains commas or blanks, the entire argument must be enclosed within angle brackets (< and >). Such arguments must still be separated from other arguments by commas. A bracketed argument with no intervening character is treated as a null argument. Embedded brackets must occur in pairs. Parameter 0 may not be bracketed and, hence, may not contain blanks (although commas are legal). Note that a macro argument may not contain the characters "<" or ">" unless they occur as part of the argument bracketing.

6.1.4 Labels Within Macros

To avoid the problem of multiply–defined labels resulting from multiple calls to a macro which employs labels in its source statements, the programmer may direct the assembler to generate unique labels on each call to a macro.

Assembler–generated labels include a string of the form .nnn, where nnn is a 3–digit value. The programmer may request an assembler–generated label by specifying \@ in a label field within a macro body. Each successive label definition which specifies a \@ directive will generate successive values of .nnn, thereby creating unique labels on repeated macro calls. Note that \@ may be preceded or succeeded by additional characters for clarity and to prevent ambiguity.

References to an assembler–generated label always refer to the label of the given form defined in the current level of the current macro expansion. Such a label is referenced as an operand by specifying the same character string as that which defines the label.

6.1.5 The MEXIT Directive

The MEXIT directive terminates the macro source statement generation during expansion. It may be used within a conditional assembly structure to skip any remaining source lines up to the ENDM directive. All conditional assembly structures pending within the macro currently being expanded are also terminated by the MEXIT directive. The MEXIT Directive takes an optional expression. It exits if the expression is true (non–zero). If the MEXIT Directive is not given an argument, comments must be delimited with a '!'.

Example

```
SAV2    MACRO
        MOVE.L   \1,SAVET        SAVE 1ST ARGUMENT
        MOVE.L   \2,SAVET+4      SAVE 2ND ARGUMENT
        IFC        3,              IS THERE A 3RD
*                                  ARGUMENT?
        FAIL     10000           DID THE ASSEMBLER
*                                    FAIL THRU HERE?
        MEXIT                    NO, EXIT FROM MACRO
        ENDC
        MOVE.L   \3,SAVET+8      SAVE 3RD ARGUMENT
ENDM
```

6.1.6 The NARG Symbol

The symbol NARG is a special symbol when referenced within a macro expansion. The value assigned to NARG is the index of the last argument passed to the macros in the parameter list (including nulls). NARG is undefined outside of macro expansion and may be referenced as a Class 1 or 2 user–defined symbol outside of a macro expansion.

6.1.7 Implementation of Macro Definition

When the sequence of source statements:

```
MAC1    MACRO
        <stmt1>
        <stmt2>
        .
        .
        .
        <stmtn>
        ENDM
```

is encountered in a source program, the following actions are performed:

1. The symbol table is checked for a Class 1 symbol entry of "MAC1". If such an entry is already present, a redefined symbol warning is generated; otherwise, an entry is placed in the symbol table, identifying MAC1 as a macro.
2. Starting with the line following the MACRO directive, each line of the macro body is saved in a character sequence identified with MAC1. In the example, stmt1 through stmtn are saved in this manner. No object code is produced at this time.
3. Normal processing resumes with the line following the ENDM directive.

6.1.8 Implementation of Macro Expansion

When the statement:

MAC1 . *<qualifier>*
<param1> , *<param2>* , . . . , *<paramn>*

is encountered in a source program calling the previously defined

macro MAC1 (above), the following actions are performed:

1. The line is scanned for parameters which are saved as literals or null values, one such value in each of the 36 parameter record fields. No object code is produced.

2. Macro expansion consists of the retrieval of the source lines which comprise the macro body. Each line is retrieved in turn, with special character pairs replaced by parameter strings or assembler–generated label strings.

   If a backslash character \ is followed by either a digit (0 through 9) or an upper case letter (A through Z), the two characters are replaced by the literal string which corresponds to that parameter on the macro invocation line(s).

   A character sequence which includes \@ is replaced by an assembler–generated label. An assembler–generated label is uniquely identified by the characters preceding and/or appended to the \@ sequence and the macro invocation in which the reference occurs. Such labels may appear anywhere in the source line and always refer to the current macro expansion.

3. When a line has been completely expanded, it is assembled as any other source input line. At this time, any errors in the syntax of the expanded assembly code are found. Expanded lines longer than 80 characters are truncated, and an error is generated.

   If a nested macro call is encountered, the nested macro expansion takes place recursively. There is no set limit to the depth of macro call nesting.

6.2 Conditional Assembly

Conditional assembly allows the programmer to write a comprehensive source program that can cover many conditions. Assembly conditions may be specified through the use of arguments in the case of macros and through definition of symbols via the SET and EQU directives. Variations of parameters can then cause assembly of only those parts necessary for the specified conditions.

The I/O section of a program, for example, will vary, depending on the target environment. Conditional assembly directives can include or exclude an I/O section, based on a flag set at the beginning of the assembly.

6.2.1 Conditional Assembly Structure

There are two conditional assembly structures available: IFC–ELSEC–ENDC and REPEATC–ENDR. IFC–ELSEC–ENDC blocks allow conditional assembly and are valid in any part of an assembly language program. REPEATC–ENDR blocks also allow conditional assembly, and are only valid within a macro definition.

The ELSEC, REPEATC, and ENDR constructs are only allowed if the "select TASKING extensions" option is in effect. See the *68000 Family Assembler* chapter in the *InterTools User's Manual* for a description of these extensions.

The IF conditional assembly structure consists of three parts:

1. *The header.* There are two conditional header clauses recognized by the assembler. The first form compares the equality of two strings:

   `IF`*xx* <*string1*>,<*string2*>

   "xx" specifies either the string compare (C) condition or the string

not compare (NC) condition, representing string equality and inequality, respectively. The result of the string comparison, along with the "xx" condition, determines whether the body of the conditional structure will be assembled. Either string may contain embedded commas or spaces. An apostrophe that occurs within a string must be specified by double apostrophes.

The second form of the conditional clause compares with an expression against zero:

```
IFxx <expression>
```

"xx" specifies a conditional relation between the expression and the value zero. The result of this comparison at assembly time determines whether the body of the conditional structure will be assembled. Valid conditional relation codes include:

EQ	expression	= 0
NE	expression	<> 0
LT	expression	< 0
LE	expression	<= 0
GT	expression	> 0
GE	expression	>= 0

Because of the nature of this comparison, the expression must be absolute. No forward references are allowed.

2. *The body.* The body of the conditional assembly structure consists of a sequence of standard source statements. There is no set limit to the depth of conditional assembly nesting; if such nesting occurs, an ENDC terminator must be specified for each structure.

   There may be an ELSE clause in the body. The keyword for for this is ELSEC.

   Note: ELSEC may only be used if the "select TASKING extensions" option is in effect. See the *68000 Family Assembler* chapter in the *InterTools User's Manual* for more information.

3. *The terminator* ENDC. When an IFxx directive is encountered, the specified condition is evaluated. If the condition is true, the statements constituting the body of the conditional assembly structure are each assembled in turn. If the relation is false, the entire conditional assembly structure is ignored; the ignored lines are not included in the assembly listing. By specifying the OPT NOCL option, the header and terminator lines are ignored for listing purposes.

   IFxx and ENDC directives may not be labeled.

The REPEATC–ENDR construct has a similar structure:

1. *The header*: REPEATC <*expr1*>[,<*expr2*>]. Both <*expr1*> and <*expr2*> must be assembly time absolute expressions. No forward references are allowed. If <*expr1*> is equal to zero (false), then statements up to the ENDR are ignored. Otherwise, the statements are assembled and the assembler repeats the process again until <*expr1*> is equal to zero. A REPEATC block stops iterating when the specified expression maximum, <*expr2*> is reached. If <*expr2*> is not specified, then the REPEATC block stops after 255 iterations.
2. *The body.* The body of a REPEATC–ENDR can contain any assembly language statements, including complete IFC–ELSEC–ENDC and REPEATC–ENDR constructs. IFC–ENDIF and REPEATC–ENDR blocks may not cross the boundary of a macro expansion or the boundaries of each other.
3. *The terminator*: ENDR. This terminates the body of the REPEATC construct.

Testing for null parameters may be done via the string compare form of the conditional assembly. To assemble conditionally if parameter 1 is null, either of the following is correct:

```
IFC "','\1'
```

or

```
IFC '\1',"
```

To assemble conditionally if a parameter is present, use either of the IFNC formats analogous to the above two.

A conditional assembly structure is also terminated by a MEXIT directive. It is an error if a conditional assembly block is not terminated in the same macro call and at the same level that it was begun in.

6.2.2 Example of Macro and Conditional Assembly Usage

The following example illustrates most of the features of macro and conditional assembly structures. The assembly code is shown as it appears, without line numbers or object code. Note that angle brackets (< >) shown in examples are required characters.

Example of Nested Macros

```
MAC0        MACRO
            MOVE.\0       \1
            CLR.L         \2
            ENDM

MAC1        MACRO
            MOVE.\        #\1,D\2
            IF\3          \1          CONDITIONAL
            ADD.\         #1,D\2
            IF\3          \1-5         NESTED CONDITIONAL
            ADD.\0        #2,D\2                    \4
            ENDC                       END NESTED
CONDITIONAL
            ENDC                       END CONDITIONAL
LAB\@       CLR.L         D1
            MOVE.\0       D\2,(A0)+
            B\3           L\@END
            BRA           LAB\@
L\@END      \5.\0         #1,D\2
            IFLE          \1
            MAC0.\0       <D\2,(A0)>,A\2 NESTED MACRO
CALL
            ENDC
            ENDM

            OPT           MEX,NOCL
            MAC1.L        7,3,GT,<TEST PASSES>,ADD
* Expansion is equivalent to following lines
            MOVE.L        #7,D3
            ADD.L         #1,D3
            ADD.L         #2,D3     TEST PASSES
```

```
LAB.001   CLR.L       D1
          MOVE.L      D3,(A0)+
          BGT         L.002END
          BRA         LAB.001
L.002END
          ADD.L       #1,D3

          MAC1.W      0,6,NE,<ERROR HERE>,SUB
* Expansion is equivalent to following lines
          MOVE.W      #0,D6
LAB.003   CLR.L       D1
          MOVE.W      D6,(A0)+
          BNE         L.004END
          BRA         LAB.003
L.004END
          SUB.W       #1,D6
          MOVE.W      D6,(A0)
          CLR.L       A6
```

Examples of REPEATC–ENDR

```
NUMSTR    MACRO
X         SET         1
          IFGT        X-9
          FAIL        "Argument to NUMSTR out of range"
          ENDC
          IFLT        X
          FAIL        "Argument to NUMSTR out of range"
          ENDC
          REPEATC     1,X
          DC.B        0+X          SAME AS X IF 0<=X<=9
X         SET         X-1
          ENDR
          DC.B        0
          ENDM

          NUMST       3

* Expansion is equivalent to following lines
          DC.B        0+3
          DC.B        0+2
          DC.B        0+1
          DC.B        0

POWERS    MACRO
X         SET         \2
          REPEATC     1,\1
          DC.\1       X
X         SET         X*\2
          ENDR
          ENDM

          POWERS.W    4,4

* Expansion is equivalent to following lines
          DC.W        4
          DC.W        16
          DC.W        64
          DC.W        256
```

CHAPTER 7

Structured Control Statements

An assembly language provides an instruction set for performing certain rudimentary operations. These operations, in turn, may be combined into control structures — such as loops (for, repeat, while) or conditional branches (if–then, if–then–else). To simplify the process of coding these constructs, this assembler accepts formal, high level directives that specify these control structures, generating, in turn, the appropriate assembly language instructions for their efficient implementation. This use of structured control statement directives improves the readability of assembly language programs, without compromising the desirable aspects of programming in an assembly language.

7.1 Keyword Symbols

The following Class 1 symbols, used in the structured syntax, are reserved keywords (directives):

ELSE	ENDW	REPEAT
ENDF	FOR	UNTIL
ENDI	IF	WHILE

The following symbols are required in the structured syntax. All keywords are reserved:

AND	DOWNTO	TO
BY	OR	
DO	THEN	

Note that AND and OR are reserved instruction mnemonics, however.

7.2 Syntax

This section describes the formats for the IF, FOR, REPEAT, and WHILE statements. They are spaced to show the line separations required for Class 1 symbol usage. Syntactic variables used in the formats are as follows:

<expression> A simple or compound expression (see the *Simple and Compound Expressions* section).

<stmtlist> Zero or more assembler directives, structured control statements, or executable instructions.

Note that an assembler directive (see the *Assembler Directives* chapter) occurring within a structured control statement is examined exactly once – at assembly time. Thus, the presence of a directive within a FOR, REPEAT, or WHILE statement does not imply repeated occurrence of an assembler directive; nor does the presence of a directive within an IF–THEN–ELSE statement imply a conditional assembly structure (see the *Structured Control Statements* chapter).

For correct recognition, the statements in <stmtlist> must not appear on the same line as the structured syntax symbols.

<size>

The value B, W, or L, indicating a data size of byte, word, or longword, respectively. With the keyword FOR, <size> is a single code applying to <op1>, <op2>, <op3>, and <op4>. With the keywords IF, UNTIL, and WHILE, <size> indicates the size of the operand comparison in the subsequent simple expression (refer to paragraph 5.3.4 for a compound expression). Note that structured syntax statements rely on the underlying opcodes and the restrictions these opcodes place on arguments to the statements. For example, the structured syntax statement

```
FOR.B D7 = #0 to #255 DO
```

generates code without warning but does not execute as expected. This is because the comparison opcode CMP does a signed comparison and hence deals with numbers in the range **–128 ... 127** instead of **0 ... 255**.

<extent>

The value S or L, indicating that the branch extent is short or long, respectively. This is appended to the keywords THEN, ELSE, and DO, to force the appropriate extent of the generated forward branch over the subsequent <stmtlist>. The default extent for the Motorola 68020–plus is determined by the forward branch option directive (OPT BRS, OPT BRB, OPT BRW, or OPT BRL) currently in effect.

<op1>

A user–defined operand whose memory/register location holds the FOR counter. **This must be a data or address register.**

<op2> The initial value of the FOR counter. The effective address may be any mode. Immediate operands must be preceded by a # sign.

<op3> The terminating value for the FOR counter. The effective address may be any mode. Immediate operands must be preceded by a # sign.

<op4> The step (increment/decrement) for the FOR counter each time through the loop. If not specified, it defaults to a value of #1. The effective address may be any mode. Immediate operands must be preceded by a # sign.

7.2.1 IF Statement

Syntax

IF[.<*size*>] <*expression*> **THEN**[.<*extent*>]
    <*stmtlist*>
ENDI

OR

IF[.<*size*>] <*expression*> **THEN**[.<*extent*>]
    <*stmtlist*>
ELSE[.<*extent*>]
    <*stmtlist*>
ENDI

If <*expression*> is true, execute the <*stmtlist*> following THEN;

If <*expression*> is false, execute <*stmtlist*> following ELSE, if present, or advance to next instruction.

Notes:

- If an operand comparison <*expression*> is specified, the condition codes are set and tested before execution of <*stmtlist*>.
- In the case of nested IF–THEN–ELSE statements, each ELSE refers to the closest IF–THEN.

Example

```
IF.L D1 <LT> #10 THEN
MOVE D5, D6
ENDI
```

7.2.2 Floating Point Structured Assembler Syntax for the IF Statement

Syntax

IF[.<*fmt*>]	**FP***n* <*fpcc*> <*ea*>	**THEN**
IF[.<*fmt*>]	<*ea*> <*fpcc*> **FP***n*	**THEN**
IF.X	**FP***n* <*fpcc*> **FP***m*	**THEN**
IF	<*fpcc*>	**THEN**

where:
<*fmt*> = B | W | L | S | D | X | P (W is default)
<*fpcc*> = A floating point condition code, as defined in Table 2–11.

This directive is similar to the non–floating point IF syntax, except that the floating point condition codes are used. When the assembler expands the structured IF statement with a floating point condition code, <*fpcc*>, it must choose the true IEEE inverse of <*fpcc*>. For example, the code generated for:

```
IF.X FP3 <FGT> #3.3 THEN
```

(where GT is one value of fpcc and #3.3 is a required constant value)

would be:

```
        FCMP.X      #3.3,FP3
        FBNGT       ELSECLAUSE
        ....                true clause code
        BRA         PAST
ELSECLAUSE
        ....                false clause code
PAST
        ....
```

Note: The branch following the FCMP is a FBNGT rather than a FBLE because FBNGT is the IEEE inverse of FBGT.

7.2.3 FOR Statement

Syntax

FOR[.<size>] <*op1*> = <*op2*> **TO** <*op3*>
[**BY** <*op4*>] **DO**[.<*extent*>] <*stmtlist*>
ENDF

OR

FOR[.<*size*>] <*op1*>=<*op2*> **DOWNTO**
<*op3*> [**BY** <*op4*>] **DO**[.<*extent*>] <*stmtlist*>
ENDF

These counting loops utilize a user–defined operand, <*op1*>, for the loop counter. FOR–TO allows counting upward, while FOR–DOWNTO allows counting downward. In both loops, the user may specify the step size, <*op4*>, or elect the default step size of #1. The FOR–TO loop is not executed if <*op2*> is greater than <*op3*> upon entry. Similarly, the FOR–DOWNTO loop is not executed if <*op2*> is less than <*op3*>.

Notes:

- The condition codes are set and tested before each execution of <*stmtlist*>. This happens even if <*stmtlist*> is not executed.
- A step size of #1 may not be meaningful if the counter, <*op1*>, is used to index through word or longword size data.

- The FOR structure generates a move, a compare, and either an add or subtract. Therefore, if any of the four operands is an address register, <*size*> may not be B (byte).
- <*op1*> must be a data or address register.

Example

```
FOR       COUNT =  #4 TO #40 BY #4 DO
          NOP      loop 10 times by steps of 4
ENDF
```

7.2.4 REPEAT Statement

Syntax

REPEAT
<*stmtlist*>
UNTIL[.<*size*>] <*expression*>

<*stmtlist*> is executed repeatedly until <*expression*> is true.

Notes:

- The <*stmtlist*> is executed at least once, even if <*expression*> is true upon entry.
- If an operand comparison <*expression*> is specified, the condition codes are set and tested following each execution of <*stmtlist*>.

Example

```
REPEAT
          MOVE      (A6)+,(A5)+
          UNTIL <EQ>
```

7.2.5 WHILE Statement

Syntax

WHILE[.<*size*>] <*expression*> **DO**[.<*extent*>]
    <*stmtlist*>
ENDW

The <*expression*> is tested before execution of <*stmtlist*>. While <*expression*> is true, <*stmtlist*> is executed repeatedly.

Notes:

- If <*expression*> is false upon entry, <*stmtlist*> is not executed.
- If an operand comparison <*expression*> is specified, the condition codes are set and tested before each execution of <*stmtlist*>. The condition codes are set and tested even if <*stmtlist*> is not executed.

Example

```
WHILE.B (A3) <NE> D2 DO
        MOVE.B       (A5)+,D3
        ENDW
```

7.3 Simple and Compound Expressions

Expressions are an integral part of IF, REPEAT, and WHILE statements. An expression may be simple or compound. A compound expression consists of no more than two simple expressions joined by AND or OR.

7.3.1 Simple Expressions

Simple expressions are concerned with the bits of the Condition Code Register (CCR). These expressions are of two types. The first type merely tests conditions currently specified by the contents of the CCR. The second type set up a comparison of two operands to set the condition codes, and afterwards tests the codes.

7.3.2 Condition Code Expressions

Fourteen tests (identical to those in the Bcc instruction) may be performed, based on the CCR condition codes. The condition codes, in this case, are preset by either a user–generated instruction or a structured operand–comparison expression. Each test is expressed in the structured control statement by a mnemonic enclosed in angle brackets (<\^>) as follows:

<CC>	<LS>
<CS>	<LT>
<EQ>	<MI>
<GE>	<NE>
<GT>	<PL>
<HI>	<VC>
<LE>	<VS>

Example

```
IF <EQ> THEN
        CLR.L     D2
ENDI

REPEAT
        SUB       D4,D3
UNTIL <LT>
```

7.3.3 Operand Comparison Expressions

Two operands may be compared in a simple expression, with subsequent transfer of control based on that comparison. Such a comparison takes the form:

<op1> <cc> <op2>

where <cc> is a condition mnemonic enclosed in angle brackets, specifying the relation to be tested between *<op1>* and *<op2>*. When processed by the assembler, this expression translates to a compare instruction. For example,

```
CMP   <op1>,<op2>
```

followed by a branch instruction (Bcc) which tests the relation specified. <op1> is normally, but not necessarily assigned to the first (leftmost) operand and <op2> to the second (rightmost) operand of the compare instruction.

Notes:

- A size may be specified for the comparison by appending a data size code (B, W, or L) to the directive, with W being the default. The only restriction is that a byte size code (B) may not be used in conjunction with an address register direct operand.
- Compare instructions require certain effective addressing modes for their operands. These modes are listed in Table 5–1. However, if the operands, *<op1>* and *<op2>*, are not listed in an order that generates a legal compare instruction (Table 5–1), but generates a legal compare if the operand order is reversed, the assembler reverses the operands when expanding the expression. To maintain the nature of the relation specified, the condition operator is adjusted, if necessary. For example,

```
D2 <GT> #5
```

is adjusted by the assembler to the equivalent of

```
#5 <LE> D2
```

Likewise,

```
A2 <EQ> (A5)
```

is adjusted to the equivalent of

```
(A5) <NE> A2.
```

This processing allows the user the flexibility of specifying the most meaningful operand order in the expression.

Table 5–1: Effective Compare Instruction Addressing Modes

	First Operand	Second Operand
CMP	(All)	Data register direct
CMPA	(All)	Address register direct
CMPI	Immediate	(Data alterable)
CMPM	Postincrement register indirect	Postincrement register indirect

If the operands, either as stated or reversed, do not yield a legal compare instruction, an error will result. For example, the statement:

```
IF (A1) <NE> (A2) THEN
```

results in an illegal address mode error during expansion. To avoid this error, a MOVE is required to effect a legal operand, such as:

```
MOVE (A2),D2 IF (A1) <NE> D2 THEN
```

Example

```
WHILE.B (A3) <NE> D2 DO          THIS EXPRESSION
        MOVE.B   (A5)+,D2        IS LEGAL AS
ENDW                             STATED.

IF D7 <LT> #10 THEN              THIS EXPRESSION
        BSR      SUBR1           IS REVERSED
ELSE
        MULS     #2,D7
ENDI
```

7.3.4 Compound Expressions

A compound expression consists of two simple expressions joined by a logical operator. The Boolean value of the compound expression is determined by the Boolean values of the simple expressions and the nature of the logical operator (AND or OR).

The two simple expressions are evaluated in the order in which they are given. However, if an AND separates the expressions and the first expression is false, the second expression is not evaluated. Likewise, if an OR separates the expressions and the first expression is true, the second expression does not need to be evaluated, and the condition codes reflect the result of only the first simple expression.

A size may be specified for each operand comparison expression. The size of the comparison for the first expression may be appended to the directive, while the size of the comparison for the second expression may be appended to the keyword AND or OR. For example, in the statement:

```
IF.L D3 <GT> (A0) OR.B #Q <EQ> BUFFER1
```

the first comparison (between D3 and (A0)) is a longword comparison, and the second (between #Q and BUFFER1) is a byte comparison.

7.4 Source Line Formatting

The format of structured source statements is more restricted than the format of basic statements. The following paragraphs discuss the formatting requirements of structured statements as well as their appearance in the assembly listing.

7.4.1 Class 1 Symbol Usage

Class 1 symbols are the assembler directives (including macro names), instruction mnemonics, and the structured control directives. Only one

of these symbols is recognized on each source line. Thus, each directive (reserved keyword) of a structured control statement and each executable instruction generated by the programmer must be written on a separate source line. The following source line, for example, is in error:

```
REPEAT MOVE (A5),D2 UNTIL <EQ>
```

The MOVE and UNTIL symbols and their operands are not recognized as class 1 symbols, but are treated as part of the comment field of the REPEAT directive. Likewise, the following lines are in error:

```
IF      <VS> THEN JSR OVERFLOW
ELSE    JMP (A3) ENDI
```

The JSR, JMP, and ENDI symbols and their operands are not recognized because they come after the THEN and Else keywords and are treated as comments. The correct format for these lines is as follows:

```
REPEAT
     MOVE    (A5),D2
UNTIL <EQ>
```

and:

```
IF <VS> THEN
     JSR     OVERFLOW
ELSE
     JMP     (A3)
ENDI
```

7.4.2 Nesting of Structured Statements

Structured statements may be nested as desired to create multilevel control structures. An example of such nesting is the following:

```
IF <EQ> THEN
     REPEAT
          MOVE     D0,(A5)+
          ADDQ     #4,D0
          MOVE.L   A4,(A4)+
     UNTIL.L A5 <LE> A4

ELSE.L

     FOR D2 = #10 TO #20 BY #2 DO
          WHILE    D4 <LE> D2 AND D4 <LT> #100
DO
                   MOVE.L  10 (A3,D4.W),(A5)+
                   ADDQ    #2,D4
          ENDW

          ENDF
     ENDI
```

Note: The indention shown above is not necessary for nested structure statements; it just makes the code easier to read.

7.5 Effects on the User's Environment

If the `-p` option on PC and Unix hosts or the `/PS` option on VMS hosts is passed on the command line, the generated code of the

structured control expansions is listed. There may be three items found in this code that will affect the user's environment:

- During assembly, local labels beginning with | (pipe bar) are generated. These labels use the same increment counter (.nnn) as local labels in macros. They are stored in the symbol table, but can not be duplicated in user–defined labels.
- In the FOR loop, <op1> is a user–defined symbol. When exiting the loop, the memory/register assigned to this symbol contains the value which caused the exit from the loop.
- Compare instructions (Table 5–1) are generated by the assembler whenever two operands are tested relationally in a structured statement. At run time, however, **these assembler–generated instructions set the condition codes of the CCR** (in the case of a loop, the condition codes are set repeatedly). Users must keep in mind the effects of this when writing code that references the CCR within or following a structured statement.

CHAPTER 8

Position–independent Code

This chapter contains sections on Forcing Position Independence, Base–Displacement Addressing, and Base–Displacement in Conjunction with Forced Position Independence.

8.1 Forcing Position Independence

When creating a relocatable program module, it is often desirable to ensure that all references to operands in relocatable sections are position–independent effective addresses, i.e., no absolute addresses occur as effective addresses for such references. To avoid absolute effective address formats, it is necessary to ensure that all memory operand references are resolved by the assembler or the linker into one

of the program counter relative or address register indirect addressing modes. The ORG directive should also be avoided.

To override an absolute address mode when resolving the effective address format of an operand, the following formats may be used to force program counter relative addressing:

- Forcing program counter with displacement:

  An operand of the form:

  ```
  LABEL(PC)
  ```

  is resolved as a PC with displacement effective address, either by the assembler or by the linker. If LABEL cannot be resolved into a 16–bit displacement from the program counter, an error is generated.

- Forcing PC with index plus displacement:

  An operand of the form:

  ```
  LABEL(PC,Rn)
  ```

  is resolved as a PC with index plus displacement effective address by the assembler. If LABEL cannot be resolved into an 8–bit displacement from the program counter, an error is generated.

8.2 Base–Displacement Addressing

Although PC relative addresses have the advantage of position independence, such address formats often are not the most meaningful to the programmer when debugging an assembled module. There are many times when a programmer would prefer to see an address relative to a specified base — i.e., in a base–displacement format. This is especially true when addressing tables, arrays, and other data structures. Base–displacement references to a given location are "base relative" and, therefore, fixed with respect to a given base address; PC relative references to that same location are different in each instruction.

Base–displacement addressing must be handled explicitly by the programmer. For example, if the following data area is declared:

```
TEMP      DS      $40
CONST       DC      $10
ARRAY1    DS.L    $10
ARRAY2    DS.L    $10
RESULT    DS.L    $10
```

the programmer may choose to load A6 with the address of TEMP and make references to the other data locations as displacements from this base address. For example, to move the first element of ARRAY1 to D1, the programmer may specify:

```
MOVE.L    ARRAY1-TEMP(A6),D1
```

Indexing with the low order contents of D0 may be added (as the array index):

```
MOVE.L    ARRAY1-TEMP(A6,D0),D1
```

8.3 Base–Displacement in Conjunction with Forced Position Independence

Complete code–position independence can be achieved by using base–displacement addressing in conjunction with the PCS option and

the forced PC relative addressing scheme outlined in the *Forcing Position Independence* section. Although these techniques can be used to avoid all undesired absolute address formats, there are significant limitations of PC relative addressing in a position–independent program, as noted below:

- *PC with displacement*:
  PC with displacement effective addresses (for the 68000 and 68010) are restricted by the 16–bit displacement field. A displacement greater than 32K byte from the current PC cannot be resolved in this format.
- *PC with index plus displacement*:
  The displacement field here is restricted to eight bits (for the 68000 and 68010), limiting the range of this format to a 128–byte displacement from the current PC. The displacement may be relocatable.
- *Operands in the alterable addressing category*:
  Neither PC relative mode is allowed as an alterable operand. This is a significant limitation in instructions which require an alterable operand, such as the destination operand in a MOVE instruction.

  By appropriate use of base registers, these limitations can be overcome.

APPENDIX A

Instruction Set Summary

This appendix provides a summary of the 68000 family instruction set. For detailed information, refer to the *M68040 32–bit Microprocessor User's Manual*, the *M68881/2 Floating Point Coprocessor User's Manual*, and the *M68851 Paged Memory Management Unit User's Manual*.

1 Conventions

The following definitions are used for the operation description in the details of the instruction set:

Operand	Definition
An	address register
Dn	data register
Rn	any data or address register
MRn	any 68030 MMU register
PRn	any 68851 MMU register
ARn	any 68EC030 ACU register
PC	program counter
SR	status register
CCR	condition codes (lower order byte of status register)
SSP	supervisor stack pointer
USP	user stack pointer
SP	active stack pointer
VAL	validate access level register
X	extend operand (from condition code)
Z	zero condition code
V	overflow condition code
Immediate Data	immediate data from the instruction
d	address displacement
Source	source contents
Destination	destination contents
Vector	location of exception vector
ea	any valid effective address
fc	function code
mask	mask value
level	level value

Subfields and Qualifiers	Definition
<bit>OF<operand>	Selects a single bit of the operand
<ea>{offset:width}	Selects a bit field
(<operand>)	The contents of the referenced location
<operand>10	The operand is binary coded decimal; operations are to be performed in decimal
<address register>)	The register
–(<address register>)	indirect operator which indicates
(<address register>) +	that the operand register points to the memory location of the instruction operand. The optional mode qualifiers are –, +, (d) and (d, ix).
#xxx or #<data>	Immediate data located with the instruction is the operand.

The following are three instruction set summary tables, one for the 68000/68010/68020–plus, one for the 68881/68882/68040/68060, and one for the 68851. The first column gives the Instruction Name Mnemonic in **boldface**. The second column gives a description of the operation, and the third column gives the Assembler Syntax.

2 68000/68010/68020–plus/CPU32 Instruction Set Summary

Mnemonic	Assembler Syntax	Operation
ABCD	ABCD Dx,Dy	Add decimal
	ABCD –(Ax),–(Ay)	with extend
ADD	ADD <ea>,Dn	Add binary (NOTE 1)
	ADD Dn<ea>	
ADDA	ADDA <ea>,An	Add address
ADDI	ADDI #<data>,<ea>	Add immediate
ADDQ	ADDQ #<data>,<ea>	Add quick
ADDX	ADDX Dx,Dy	Add extended
	ADDX –(Ax),–(Ay)	
AND	AND Dn,<ea>	AND logical
ANDI	ANDI #<data>,<ea>	AND immediate
ANDI to CCR	ANDI #<data>,CCR	AND immediate to condition codes
ANDI to SR	ANDI #<data>,SR	AND immediate to status register
ASL, ASR	ASd Dx,Dy	Arithmetic shift
	ASd #<data>,Dy	ASd <ea>
Bcc	Bcc <label>	Branch conditionally
BCHG	BCHG Dn,<ea>	Test a bit
	BCHG #<data>,<ea>	and change
BCLR	BCLR Dn,<ea>	Test a bit
	BCLR#<dat>,<ea>	and clear

Mnemonic	Assembler Syntax	Operation
BFCHG	BFCHG <ea> {<offset>:<width>}	Complement bit field (68020–plus)
BFCLR	BFCLR<ea> {<offset>:<width>}	Clear bit field (68020–plus)
BFEXTS	BFEXTS<ea> {<offset>:<width>},Dn	Extract bit field signed (68020–plus)
BFEXTU	BFEXTU <ea> {<offset>:<width>},Dn	Extract bit field unsigned (68020–plus)
BFFFO	BFFO <ea> {<offset>:<width>},Dn	Find first one in bit field (68020–plus)
BFINS	BFINS Dn,<ea> {<offset>:<width>}	Insert bit field (68020–plus)
BFSET	BFSET <ea> {<offset>:<width>},Dn	Set bit field (68020–plus)
BFTST	BFTST <ea> {<offset>:<width>}	Test bit field (68020–plus)
BGND	BGND	Background mode (CPU32)
BKPT	BKPT #<vector>	Breakpoint (except 68000)
BRA	BRA <label>	Branch always
BSET	BSET Dn,<ea> BSET #<data>,<ea>	Test a bit and set
BSR	BSR <label>	Branch to subroutine
BTST	BTST Dn,<ea> BTST #<data>,<ea>	Test a bit
CALLM	CALLM #ddd,<ea>	Call module (68020)
CAS	CAS Dc,Du,<ea>	Compare and swap with operand (68020–plus)

Mnemonic	Assembler Syntax	Operation
CAS2	CAS2 Dc1:Dc2, Du1:Du2,(Rn1):(Rn2)	Compare and swap with operand (68020–plus)
CHK	CHK <ea>,Dn	Check register against bounds
CHK2	CHK2 <ea>,Rn	Check register against bounds (68020–plus/CPU32)
CINV	CINVL <caches>,(An) CINVP <caches>,(An) CINVA <caches>	Invalidate cache lines (68040/68EC040/68060) (NOTE 6)
CLR	CLR <ea>	Clear an operand
CMP	CMP <ea>,Dn	Arithmetic compare
CMPA	CMPA <ea>,An	Arithmetic compare address
CMPI	CMPI #<data>,<ea>	Compare immediate
CMPM	CMPM (Ax)+,(Ay)+	Compare memory
CMP2	CMP2<ea>,Rn	Compare register against bounds (68020–plus/CPU32)
CPUSH	CPUSHL <caches>,(An) CPUSHP <caches>,(An) CPUSHA <caches>	Push and invalidate cache lines (68040/68EC040/68060) (NOTE 6)
DBcc	DBcc Dn,<label>	Test condition and decrement and branch (NOTE 2)
DIVS	DIVS <ea>,Dn	Signed divide
DIVSL	DIVSL <ea>,Dr:Dq	Signed long divide (68020–plus/CPU32)
DIVU	DIVU <ea>,Dn	Unsigned divide
DIVUL	DIVUL <ea>,Dr:Dq	Unsigned long divide (68020–plus/CPU32)

Mnemonic	Assembler Syntax	Operation
EOR	EOR Dn,<ea>	Exclusive OR logical
EORI	EORI #<data>,<ea>	Exclusive OR immediate
EORI to CCR	EORI #<data>,CCR	Exclusive OR immediate to condition codes
EORI to SR	EORI #<data>,SR	Exclusive OR immediate to status register
EXG	EXG Rx,Ry	Exchange registers
EXT	EXT Dn	Sign extend
EXTB	EXTB Dn	Sign extend byte (68020–plus/CPU32)
JMP	JMP <ea>	Jump
JSR	JSR <ea>	Jump to subroutine
LEA	LEA <ea>,An	Load effective address
LINK	LINK An,#<disp>	Link and allocate (NOTE 5)
LPSTOP	LPSTOP #<data>	Low power STOP (CPU32/68060)
LSL,LSR	LSd Dx,Dy LSd #<data>,DyS LSd <ea>	Logical shift
MOVE	MOVE <ea>,<ea>	Move data from source to destination
MOVE to SR	MOVE <ea>,SR	Move to the status register
MOVE from SR	MOVE SR,<ea>	Move from the status register

Mnemonic	Assembler Syntax	Operation
MOVE to CCR	MOVE <ea>,CCR	Move to condition codes

Mnemonic	Assembler Syntax	Operation
MOVE from CCR	MOVE CCR,<ea>	Move from condition codes (except 68000)
MOVE USP	MOVE USP,An	Move user stack pointer
	MOVE An,USP	
MOVEA	MOVEA <ea>,An	Move address
MOVEC	MOVEC Rc,Rn	Move to/from control register (except 68000) (NOTE 3)
	MOVEC Rn,Rc	
MOVEM	MOVEM <register list>,<ea>	Move multiple registers (NOTE 4)
	MOVEM <ea>,<register list>	
MOVEP	MOVEP Dx,d(Ay)	Move peripheral data
	MOVEP d(Ax),Dy	
MOVEQ	MOVEQ #<data>,Dn	Move quick
MOVES	MOVES <ea>,Rn	Move to/from address address (except 68000)
	MOVES Rn,<ea>	
MOVE16	MOVE16 (Ax)+,(Ay)+	Move 16 bytes block (68040/68EC040/68060)
	MOVE16 xxx.L,(An)	
	MOVE16 xxx.L,(An)+	
	MOVE16 (An),xxx.L	
	MOVE16 (An)+,xxx.L	
MULS	MULS <ea>,Dn	Signed multiply
MULU	MULU <ea>,Dn	Unsigned multiply
NBCD	NBCD <ea>	Negate decimal with extend
NEG	NEG <ea>	2's complement negation
NEGX	NEGX <ea>	Negate with extend
NOP	NOP	No operation

Mnemonic	Assembler Syntax	Operation
NOT	NOT <ea>	Logical complement

Mnemonic	Assembler Syntax	Operation
OR	OR <ea>,Dn	Inclusive OR
	OR Dn,<ea>	logical
ORI	ORI #<data>,<ea>	Inclusive OR immediate
ORI to CCR	ORI #<data>,CCR	Inclusive OR immediate to condition codes
ORI to SR	ORI #<data>,SR	Inclusive OR immediate to status register
PACK	PACK –(Ax),–(Ay)	Pack BCD
	PACK Dx,Dy	(68020–plus)
PEA	PEA <ea>	Push effective address
PFLUSH	PFLUSHA	Flush entry in the ATC
	PFLUSH <fc>,#<mask>	(68030)
	PFLUSH <fc>,#<mask>,<ea>	
PFLUSH	PFLUSH (An)	Flush entry at the ATC
	PFLUSHN (An)	(68040/68060)
	PFLUSHA	
PLPA	PLPAR (An)	Load Physical Address
	PLPAN (An)	PAN (68040)
	PFLUSHAN	
PLOAD	PLOADR <fc>,<ea>	Load an entry into
	PLOADW <fc>,<ea>	the ATC (68030)
PMOVE	PMOVE MRn,<ea>	Move to/from MMU
	PMOVE <ea>,MRn	registers (68030)
	PMOVEFD <ea>,MRn	
PMOVE	PMOVE ARn,<ea>	Move to/from ACU registers
	PMOVE <ea>,ARn	(68EC030)

Mnemonic	Assembler Syntax	Operation
PTEST	PTESTR <fc>,<ea>, #<level>	Test a logical address (68030)
	PTESTR <fc>,<ea>, #<level>,An	
	PTESTW <fc>,<ea>, #<level>	
	PTESTW <fc>,<ea>, #<level>,An	
PTEST	PTESTR <fc>,<ea>,	Test a logical address
	PTESTW <fc>,<ea>,	(68EC030)
PTEST	PTESTR (An)	Test a logical address
	PTESTW (An)	(68040/68EC040)
RESET	RESET	Reset external devices
ROL, ROR	ROd Dx,Dy	Rotate without extend
	ROd #<data>,Dy	ROd <ea>
ROXL, ROXR	ROXd Dx,Dy	Rotate with extend
	ROXd #<data>,Dy	ROXd <ea>
RTD	RTD #<disp>	Return from subroutine with displacement (except 68000) (NOTE 5)
RTE	RTE	Return from exception
RTM	RTM Rn	Return from module (68020)
RTR	RTR	Return and restore condition codes
RTS	RTS	Return from subroutine
SBCD	SBCD Dx,Dy	Subtract decimal
	SBCD –(Ax),–(Ay)	with extend
Scc	Scc <ea>	Set according to condition

Mnemonic	Assembler Syntax	Operation
STOP	STOP #<data>	Stop program executive
SUB	SUB <ea>,Dn	Subtract binary
	SUB Dn,<ea>	
SUBA	SUBA <ea>,An	Subtract address
SUBI	SUBI #<data>,<ea>	Subtract immediate
SUBQ	SUBQ #<data>,<ea>	Subtract quick
SUBX	SUBX Dx,Dy	Subtract with extend
	SUBX –(Ax),–(Ay)	
SWAP	SWAP Dn	Swap register halves
TAS	TAS <ea>	Test and set an operand
TBLS	TBLS.<size> <ea>,Dx	Table lookup and
	TBLS.<size>,Dym:Dyn,Dx	interpolate (signed),
		result rounded (CPU32)
TBLSN	TBLSN.<size> <ea>,Dx	Table lookup and
	TBLSN.<size>,Dym:Dyn,Dx	interpolate (signed),
		result not rounded (CPU32)
TBLU	TBLU.<size> <ea>,Dx	Table lookup and
	TBLU.<size>,Dym:Dyn,Dx	interpolate (unsigned),
		result rounded (CPU32)
TBLUN	TBLUN.<size> <ea>,Dx	Table lookup and
	TBLUN.<size>,Dym:Dyn,Dx	interpolate (unsigned),
		result not rounded (CPU32)
TRAP	TRAP #<vector>	Trap
TRAPcc	TRAPcc	Trap on condition
	TRAPcc #xxx	code (68020–plus/CPU32)
TRAPV	TRAPV	Trap on overflow
TST	TST <ea>	Test an operand

Mnemonic	Assembler Syntax	Operation
UNLK	UNLK An	Unlink
UNPK	UNPK –(Ax),–(Ay)	Unpack BCD
	UNPK Dx,Dy	(68020–plus)

Notes:

- <ea> specifies effective address.
- The assembler accepts DBRA for the F (never true) condition.
- Rc specifies control register.
- <register list> specifies the registers selected for transfer to or from memory. <register list> may be:

  Rn: a single register.

  Rn–Rm: a range of consecutive registers with m being greater than n.

  Any combination of the above, separated by a slash.
- <disp> is a 2's complement integer, 16 bits in size, which is sign extended to 32 bits before adding to the stack pointer.
- <caches> specifies cache lines.

3 68881/68882/68040/68060 Floating Point Instruction Set Summary

Mnemonic	Assembler Syntax	Operation
FABS	FABS <ea>,FPn	Absolute value function
	FABS FPm,FPn	

Mnemonic	Assembler Syntax	Operation
	FABS FPn	
	* FrABS <ea>,FPn	
	* FrABS FPm,FPn	
	* FrABS FPn	
FACOS **	FACOS <ea>,FPn	Arccosine function
	FACOS FPM,FPn	
	FACOS FPn	
FADD	FADD <ea>,FPn	Floating point add
	FADD FPm,FPn	
	* FrADD <ea>,FPn	
	* FrADD FPm,FPn	
FASIN **	FASIN <ea>,FPn	Arcsine function
	FASIN FPm,FPn	
	FASIN FPn	
FATAN **	FATAN <ea>,FPn	Arctangent function
	FATAN FPm,FPn	
FATANH **	FATANH<ea>,FPn	Hyperbolic arctangent function
	FATANH FPm,FPn	
	FATANH FPn	
FBfpcc	FBfpcc <label>	Coprocessor branch conditionally

Mnemonic	Assembler Syntax	Operation
FCMP	FCMP <ea>,FPn	Floating point compare
	FCMP FPm,FPn	
FCOS **	FCOS <ea>,FPn	Cosine function
	FCOS FPm,FPn	
	FCOS FPn	
FCOSH **	FCOSH <ea>,FPN	Hyperbolic cosine
	FCOSH FPm,FPn	function
	FCOSH FPn	
FDBfpcc	FDBfpcc DN,<label>	Decrement and branch on condition
FDIV	FDIV <ea>,FPn	Floating point divide
	FDIV FPm,FPn	
	* FrDIV <ea>,FPn	
	* FrDIV FPm,FPn	
FETOX **	FETOX <ea>,FPn	e**x function
	FETOX FPm,FPn	
	FETOX FPn	
FETOXM1 **	FETOXM1 <ea>,FPn	e**x(x–1) function
	FETOXM1 FPm,FPn	
	FETOXM1 FPn	
FGETEXP **	FGETEXP <ea>,FPn	Get the exponent function
	FGETEXP FPm,FPn	
	FGETEXP FPn	
FGETMAN **	FGETMAN <ea>,FPn	Get the Mantissa function
	FGETMAN FPm,FPn	
	FGETMAN FPn	

Mnemonic	Assembler Syntax	Operation
FINT **	FINT <ea>,FPn	Integer part function
	FINT FPm,FPn	
	FINT FPn	
FINTRZ **	FINTRZ <ea>,FPn	Integer part, rount–to–zero
	FINTRZ FPm,FPn	function
	FINTRZ FPn	
FLOG10 **	FLOG10 <ea>,FPn	Common log function
	FLOG10 FPm,FPn	
	FLOG10 FPn	
FLOG2 **	FLOG2 <ea>,FPn	Binary log function
	FLOG2 FPm,FPn	
	FLOG2 FPn	
FLOGN **	FLOGN <ea>,FPn	Natural log function
	FLOGN FPm,FPn	
	FLOGN FPn	
FLOGNP1 **	FLOGNP1 <ea>,FPn	Natural log (X+1) function
	FLOGNP1 FPm,FPn	
	FLOGNP1 FPn	
FMOD **	FMOD <ea>,FPn	Modulus function
	FMOD FPm,FPn	

Mnemonic	Assembler Syntax	Operation
FMOVE	FMOVE <ea>,FPn FMOVE FPm,FPn	Move to floating point register from memory or another floating point register
	FMOVE FPN,<ea> FMOVE.P FPn,<ea>{#k} FMOVE.P FPn,<ea>{Dn}	Move from floating point register to memory
	FMOVE <ea>, CONTROL | STATUS | IADDR, FMOVE CONTROL | STATUS | IADDR,<ea> * FrMOVE <ea>,FPn	Move to/from memory from/to special register
FMOVECR	FMOVECR #ccc,FPn	Move a ROM–stored constant to a floating point register
FMOVEM	FMOVEM<ea>, <fp_reg_list>	Move to multiple floating point registers
	FMOVEM <ea>,Dn FMOVEM <ea>, CONTROL | STATUS | IADDR	Move to a data register or special register
	FMOVEM <fp_reg_list>,<ea>	Move from multiple floating point registers to memory
	FMOVEM Dn,<ea> FMOVEM CONTROL | STATUS | IADDR,<ea>	Move from data register or special register to memory

Mnemonic	Assembler Syntax	Operation
FMUL	FMUL <ea>,FPn	Floating point multiply
	FMUL FPm,FPn	
	* FrMUL <ea>,FPn	
	* FrMUL FPm,FPn	
FNEG	FNEG <ea>,FPn	Negate function
	FNEG FPm,FPn	
	FNEG FPn	
	* FrNEG <ea>,FPn	
	* FrNEG FPm,FPn	
	* FrNEG FPn	
FNOP	FNOP	Floating point NO–OP
FREM **	FREM <ea>,FPn	Floating point remainder
	FREM FPM,FPn	
FRESTORE	FRESTORE <ea>	Restore internal state of coprocessor
FSAVE	FSAVE <ea>	Coprocessor save
FSCALE **	FSCALE <ea>,FPn	Floating point scale exponent
	FSCALE FPm,FPn	
FSfpcc	FSfpcc <ea>	Set on condition (68881/68882/68040)
FSGLDIV **	FSGLDIV <ea>,FPn	Floating point single
	FSGLDIV FPm,FPn	precision divide
FSGLMUL **	FSGLMUL <ea>,FPn	Floating point single
	FSGLMUL FPm,FPn	precision multiply
FSIN **	FSIN <ea>,FPn	Sine function
	FSIN FPm,FPn	
	FSIN FPn	
FSINCOS **	FSINCOS <ea>,FPm:FPn	Sine–cosine function

Mnemonic	Assembler Syntax	Operation
FSINH **	FSINH <ea>,FPn	Hyperbolic sine function
	FSINH FPm,FPn	
	FSINH FPn	
FSQRT	FSQRT <ea>,FPn	Square root function
	FSQRT FPm,FPn	
	FSQRT FPn	
	* FrSQRT <ea>,FPn	
	* FrSQRT FPm,FPn	
	* FrSQRT FPn	
FSUB	FSUB <ea>,FPn	Floating point subtract
	FSUB FPm,FPn	
	* FrSUB <ea>,FPn	
	* FrSUB FPm,FPn	
FTAN **	FTAN <ea>,FPn	Tangent function
	FTAN FPm,FPn	
	FTAN FPn	
FTANH **	FTANH <ea>,FPn	Hyperbolic tangent function
	FTANH FPm,FPn	
	FTANH FPn	
FTENTOX **	FTENTOX <ea>,FPn	10**x function
	FTENTOX FPm,FPn	
	FTENTOX FPn	
FTRAPfpcc	FTRAPfpcc	Trap on condition
	FTRAPfpcc #xxx	(68881/68882/68040)
FTST	FTST <ea>	Floating point test an operand
FTWOTOX **	FTWOTOX <ea>,FPn	2**x function

Mnemonic	Assembler Syntax	Operation
	FTWOTOX FPm,FPn FTWOTOX FPn .TE .bp	

Notes:

* Supported by 68040/68060 only; r is the precision (S for single or D for double).

** On the 68040/68060, these instructions trap as an F–line exception and must be emulated in software by the F–line exception handler.

4 68851 Instruction Set Summary

Mnemonic	Assembler Syntax	Operation
PBpcc	PBpcc <label>	Branch on PMMU condition
PDBpcc	PDBpcc Dn,<label>	Test, decrement, and branch on condition
PFLUSH	PFLUSHA PFLUSH <fc>,#<mask> PFLUSH <fc>,#<mask>,<ea> PFLUSHS <fc>,#<mask> PFLUSHS <fc>,#<mask>,<ea>	Invalidate entries in the ATC
PFLUSHR	PFLUSHR <ea>	Invalidate ATC and RPT entries
PLOAD	PLOADR <fc>,<ea> PLOADW <fc>,<ea>	Load an entry into the ATC
PMOVE	PMOVE PRn,<ea>	Move PMMU register

Mnemonic	Assembler Syntax	Operation
	PMOVE <ea>,PRn	
PRESTORE	PRESTORE <ea>	PMMU restore function
PSAVE	PSAVE <ea>	PMMU save function
PSpcc	PSpcc <ea>	Set on PMMU condition
PTEST	PTESTR <fc>,<ea>,#<level>	Test a logical address
	PTESTR <fc>,<ea>, #<level>,An	
	PTESTW <fc>,<ea>,#<level>	
	PTESTW <fc>,<ea>, #<level>,An	
PTRAPpcc	PTRAPpcc	Trap on PMMU condition
	PTRAPpcc #xxx	
PVALID	PVALID VAL,<ea>	Validate a pointer
	PVALID An,<ea>	

APPENDIX B

Character Set

This appendix lists the ASCII characters recognized by the assembler.

1 Characters Recognized

The characters recognized by the assembler are listed below. The ASCII codes for these characters are shown on the following pages:

- The upper case letters A through Z
- The lower case letters a through z
- The digits 0 through 9
- Five arithmetic operators: +, –, *, /, and %
- The logical operators: >>, <<, &, and !
- Parentheses used in expressions ( )
- Characters used as special prefixes:
  # (pound sign) specifies the immediate mode of addressing
  $ (dollar sign) specifies a hexadecimal number
  @ (commercial "at") specifies an octal number
  % (percent) specifies a binary number
  ' (apostrophe) specifies an ASCII literal character
- The special characters used in macros: <, >, /, and @
- Four separating characters:
  (space)
  (tab)

, (comma)
. (period)

- A comment in a source statement may include any characters with ASCII values from (hexadecimal) 20 through 7E.
- Character used as a special suffix:
  : (colon) specifies the end of a label

2 ASCII Character Set

CHARACTER	COMMENTS	HEX VALUE
NUL	Null or tape feed	00
SOH	Start of Heading	01
STX	Start of Text	02
ETX	End of Text	03
EOT	End of Transmission	04
ENQ	Enquire	05
ACK	Acknowledge	06
BEL	Bell	07
BS	Backspace	08
HT	Horizontal Tab	09
LF	Line Feed	0A
VT	Vertical Tab	0B
FF	Form Feed	0C
RETURN	Carriage Return	0D
SO	Shift Out (to red ribbon)	0E
SI	Shift In (to black ribbon)	0F
DLE	Data Link Escape	10

CHARACTER	COMMENTS	HEX VALUE
DC1	Device Control 1	11
DC2	Device Control 2	12
DC3	Device Control 3	13
DC4	Device Control 4	14
NAK	Negative Acknowledge	15
SYN	Synchronous idle	16
ETB	End of Transmission Block	17
CAN	Cancel	18
EM	End of Medium	19
SUB	Substitute	1A
ESC	Escape, prefix	1B
FS	File Separator	1C
GS	Group Separator	1D
RS	Record Separator	1E
US	Unit Separator	1F
SP	Space or blank	20
!	Exclamation point	21
"	Quotation mark	22
#	Number sign	23
$	Dollar sign	24
%	Percent sign	25
&	Ampersand	26
'	Apostrophe	27
(	Opening parenthesis	28

CHARACTER	COMMENTS	HEX VALUE
)	Closing parenthesis	29
*	Asterisk	2A
+	Plus sign	2B
–	Hyphen (minus)	2D
.	Period (decimal point)	2E
/	Slant	2F
0	Digit 0	30
1	Digit 1	31
2	Digit 2	32
3	Digit 3	33
4	Digit 4	34
5	Digit 5	35
6	Digit 6	36
7	Digit 7	37
8	Digit 8	38
9	Digit 9	39
:	Colon	3A
;	Semicolon	3B
<	Less than	3C
=	Equals	3D
>	Greater than	3E
?	Question mark	3F
@	Commercial at	40
A	Upper case letter A	41

CHARACTER	COMMENTS	HEX VALUE
B	Upper case letter B	42
C	Upper case letter C	43
D	Upper case letter D	44
E	Upper case letter E	45
F	Upper case letter F	46
G	Upper case letter G	47
H	Upper case letter H	48
I	Upper case letter I	49
J	Upper case letter J	4A
K	Upper case letter K	4B
L	Upper case letter L	4C
M	Upper case letter M	4D
N	Upper case letter N	4E
O	Upper case letter O	4F
P	Upper case letter P	50
Q	Upper case letter Q	51
R	Upper case letter R	52
S	Upper case letter S	53
T	Upper case letter T	54
U	Upper case letter U	55
V	Upper case letter V	56
W	Upper case letter W	57
X	Upper case letter X	58
Y	Upper case letter Y	59

CHARACTER	COMMENTS	HEX VALUE
Z	Upper case letter Z	5A
[	Opening bracket	5B
\	Reverse slant	5C
]	Closing bracket	5D
^	Circumflex	5E
–	Underline	5F
'	Quotation mark	60
a	Lower case letter a	61
b	Lower case letter b	62
c	Lower case letter c	63
d	Lower case letter d	64
e	Lower case letter e	65
f	Lower case letter f	66
g	Lower case letter g	67
h	Lower case letter h	68
i	Lower case letter i	69
j	Lower case letter j	6A
k	Lower case letter k	6B
l	Lower case letter l	6C
m	Lower case letter m	6D
n	Lower case letter n	6E
o	Lower case letter o	6F
p	Lower case letter p	70
q	Lower case letter q	71

CHARACTER	COMMENTS	HEX VALUE
r	Lower case letter r	72
s	Lower case letter s	73
t	Lower case letter t	74
u	Lower case letter u	75
v	Lower case letter v	76
w	Lower case letter w	77
x	Lower case letter x	78
y	Lower case letter y	79
z	Lower case letter z	7A
{	Opening brace	7B
|	Vertical line	7C
}	Closing brace	7D
~	Equivalent	7E
DEL	Delete	7F

Index

Symbols

_tolower, 2–33

_toupper, 2–34

Numbers

68000, design module, 2–2

68302, 2–4, 2–11

68340, 2–4, 2–12

68360, 2–13

68881, floating point, 2–6, 2–15

A

A5 register, 2–5

A7 register, 2–5

abort, 2–35

abs, 2–36

acos, 2–37

address modes, 4–31–4–66
  addressing summary, 4–58
  direct address, 4–36
  effective addressing mode formats, 4–37–4–66
  errors
    displacement range violations, 4–60
    inappropriate absolute short address, 4–60
  immediate, 4–36
  memory direct, 4–34
  PC–relative address, 4–35
  register direct, 4–34, 4–37–4–66

address ranges, special address ranges, 4–51

address translation cache
  flush entries (PFLUSH), 4–102, 4–103
  invalidate (PFLUSH/PFLUSHA/PFLUSHS), 4–132
  invalidate with root pointer table entries (PFLUSHR), 4–133
  load (PLOAD), 4–104
  load (PLOADR/PLOADW), 4–134

asctime, 2–38

asin, 2–39

assert, 2–40

atan, 2–41

atan2, 2–42

atanh, 2–43

atexit, 2–44

atof, 2–45

atoi, 2–46

atol, 2–47

B

bit fields, 4–89–4–138
  double operand, 4–93–4–138
    extract, signed (BFEXTS), 4–93–4–94
    extract, unsigned (BFEXTU), 4–93–4–94
    find first one in (BFFFO), 4–94–4–95
    insert (BFINS), 4–95–4–96
  single operand, 4–89–4–138
    clear (BFCLR), 4–91–4–92
    complement (BFCHG), 4–89–4–91
    set (BFSET), 4–91–4–92
    test (BFTST), 4–92–4–93
bounds check instructions (CHK2/CMP2), 4–96
breakpoint (BKPT), 4–101
bsearch, 2–48

C

cache lines
  invalidate (CINV), 4–109
  push and invalidate (CPUSH), 4–110
calloc, 2–49
ceil, 2–50
clearerr, 2–51
clock, 2–52
compare and swap with operand (CAS), 4–100
condition codes, paged memory management unit (68851), 4–130
controls
  external symbol, 5–29–5–32
    external symbol definition (XDEF), 5–30–5–32
    external symbol reference (XREF), 5–30–5–32
    relocatable identification record (IDNT), 5–29–5–32
  internal assembly, 5–31–5–32
    declare external symbol (_BRINGIN), 5–32
    define data group (_DGROUP), 5–32
    put out debugging information (_DEBSYM), 5–32
conventions, documentation, 1–4–1–8
coprocessor state
  restore (FRESTORE), 4–126
  save (FSAVE), 4–125
cos, 2–53
cosh, 2–54
ctime, 2–55

D

data initialization, 2–141
decrement and branch on condition (DB), 4–80
difftime, 2–56
directives
  assembly control, 5–2–5–32
    absolute origin (ORG), 5–7–5–32

define offsets (OFFSET), 5–6–5–32
enter named common section (COMMON), 5–2–5–32
include secondary file (INCLUDE), 5–5–5–32
program end (END), 5–4–5–32
relocatable ORG (RORG), 5–10–5–32
relocatable program section (SECTION), 5–12–5–32
reserve storage (RESERVE), 5–8–5–32
resume defined section (RESUME), 5–9–5–32
data definition/storage allocation, 5–16–5–32
define constant (DC), 5–17–5–32
define constant block (DCB), 5–20–5–32
define storage (DS), 5–21–5–32
specify command line (COMLINE), 5–17–5–32
symbol definition, 5–13–5–32
define register list (REG), 5–15–5–32
equate floating point symbol value (FEQU), 5–14–5–32
equate symbol value (EQU), 5–14–5–32
set symbol value (SET), 5–16–5–32
terminate macro source statement generation (MEXIT), 6–6

div, 2–57

documentation, 1–1–1–8
conventions, 1–4–1–8

E

exit, 2–58

exp, 2–59

expressions
absolute, 4–29, 4–37
complex relocatable, 4–29, 4–37
simple relocatable, 4–29, 4–37

F

fabs, 2–60

fclose, 2–61

feof, 2–62

ferror, 2–63

fflush, 2–64

fgetc, 2–65

fgetpos, 2–66

fgets, 2–67

file control block, 2–7

floating point
arithmetic operations (F), 4–116
compare (FCMP/FTST), 4–121
condition codes, 4–114–4–138
conditional branch (FB), 4–122
condtion codes, 68881/68882/68040–specific, 4–115–4–138
constant notation, 4–13
decrement and branch on condition (FDB), 4–123
functions (F), 4–126
hex constant notation, 4–13–4–14
move multiple, 4–120

move register to memory (FMOVE), 4–119
move to register (FMOVE/FDMOVE/FSMOVE), 4–117
NO–OP (FNOP), 4–129
set on condition (FS), 4–124
trap on condition (FTRAP), 4–124
floating point compare (FCMP/FTST), 4–121
floor, 2–68
fmod, 2–69
fopen, 2–70
fprintf, 2–71
fputc, 2–72
fputs, 2–73
fread, 2–74
free, 2–75
freopen, 2–76
frexp, 2–77
fscanf, 2–78
fseek, 2–79
fsetpos, 2–80
ftell, 2–81
functions
floating point (F), 4–126–4–138
sine/cosine (FSINCOS), 4–128–4–138
fwrite, 2–82

G

get information about logical address (PTESTR/PTESTW), 4–136
getc, 2–83
getchar, 2–84
getenv, 2–85
getl, 2–86
gets, 2–87
getw, 2–88
global
data, 2–5
variable, 2–8
gmtime, 2–89
grouping characters, 4–33

I

I/O, system, 2–7–2–10
idata, 2–5
initialization, routine, 2–14
instructions
CHK (check), 4–71
CMP (compare), 4–71
MOVE, 4–70
isalnum, 2–90
isalpha, 2–91
iscntrl, 2–92
isdigit, 2–93
isgraph, 2–94
islower, 2–95

isprint, 2–96
ispunct, 2–97
isspace, 2–98
isupper, 2–99
isxdigit, 2–100

L

labs, 2–101
ldata, 2–5
ldexp, 2–102
ldiv, 2–103
libraries that do not use A5, 2–5
library, modification, 2–1
local effective address (LEA), 4–83
localeconv, 2–104
localtime, 2–105
log, 2–106
log10, 2–108
log2, 2–107
longjmp, 2–109

M

M68302ADS, 2–2, 2–4, 2–11
M68340BCC, 2–2, 2–4, 2–12
M68360QUADS, 2–13
macro calls, 4–4
MACSBUG, 2–2
malloc, 2–110, 2–111, 2–112, 2–113
memccpy, 2–114
memchr, 2–115
memcmp, 2–116
memcpy, 2–117
memmove, 2–118
memset, 2–119
mktime, 2–120
mnemonics
  directive, 4–4
  instruction, 4–4, 4–67–4–88
modf, 2–121
module instructions
  call module (CALLM), 4–98
  return from module (RTM), 4–98
move
  16 bytes block (MOVE16), 4–110
  floating point move multiple (FMOVEM), 4–120
  floating point register to memory (FMOVE), 4–119
  move instruction (MOVE), 4–70
  multiple registers (MOVEM), 4–81
  PMMU register (PMOVE), 4–135
  to floating point register (FMOVE/FDMOVE/FSMOVE), 4–117
  to/from ACU registers (PMOVE), 4–105
  to/from address space (MOVES), 4–88
  to/from control register (MOVEC), 4–84
  to/from MMU registers (PMOVE), 4–105

N

no−floats library, 2−17

null displacements, 4−33

O

offsetof, 2−122

omitted registers and displacements, 4−33

operands
  external reference and instruction ORG, 4−66
  external reference and instruction SECTION, 4−65–4−66
  location of operand and instruction follows, 4−63–4−66
  location of operand and instruction sections, 4−62–4−66
  operand resolution, 4−61–4−62
  unknown location of operands and instructions, 4−64–4−66

operations
  arithmetic
    ADD, 4−67
    DIV (divide), 4−69
    MUL (multiply), 4−68
    SUB (subtract), 4−67
  bit, 4−75–4−88
  branch, 4−76
  floating point arithmetic (F), 4−116
  jump, 4−79
  logical, NOT, 4−73
  shift, 4−73–4−88

operators
  arithmetic operators, 4−26
  logical operators, 4−26
  shift operators, 4−26

options, listing control and output, 5−22–5−32
  assembler option (OPT), 5−23–5−32
  control listing file format (FORMAT/NOFORMAT), 5−22–5−32
  control listing generation (LIST/NOLIST), 5−23–5−32
  control pagination (PAGE/NOPAGE), 5−27–5−32
  produce no object module (NOOBJ), 5−23–5−32
  programmer generated error (FAIL), 5−22–5−32
  set subtitle (STTL), 5−28–5−32
  set title (TTL), 5−29–5−32
  space between source lines (SPC), 5−28–5−32
  specify listing line length (LLEN), 5−23–5−32

P

packed decimal
  PACK, 4−97
  UNPK (unpack), 4−97

perror, 2−123

PMMU condition
  branch on (PB), 4−130
  move register (PMOVE), 4−135
  restore function (PRESTORE), 4−135
  save function (PSAVE), 4−136

set on (PS), 4–131
test, decrement, and branch on (PDB), 4–131
trap on (PTRAP), 4–132
pow, 2–124
printf, 2–125–2–132
putc, 2–133
putchar, 2–134
putl, 2–135
puts, 2–136
putw, 2–137

Q

qsort, 2–138

R

raise, 2–139
rand, 2–140
rcopy, 2–141
realloc, 2–142
remove, 2–143
rename, 2–144, 2–145
ROM, 2–183
run time library, 2–1–2–6
index file, 2–17
modification, 2–1, 2–14–2–32
object modules, 2–17–2–32
routines, 2–18–2–32
run–time library, routines, source code, 2–2
runtime library routine
_tolower, 2–33
_toupper, 2–34
abort, 2–35
abs, 2–36
acos, 2–37–2–204
asctime, 2–38
asin, 2–39
assert, 2–40
atan, 2–41
atan2, 2–42
atanh, 2–43
atexit, 2–44
atof, 2–45
atoi, 2–46
atol, 2–47
bsearch, 2–48
calloc, 2–49
ceil, 2–50
clearerr, 2–51
clock, 2–52
cos, 2–53
cosh, 2–54
ctime, 2–55
difftime, 2–56
div, 2–57
exit, 2–58
exp, 2–59
fabs, 2–60
fclose, 2–61
feof, 2–62
ferror, 2–63
fflush, 2–64
fgetc, 2–65, 2–83
fgetpos, 2–66
fgets, 2–67
floor, 2–68
fmod, 2–69
fopen, 2–70
fprintf, 2–71
fputc, 2–72

fputs, 2–73
fread, 2–74
free, 2–75
freopen, 2–76
frexp, 2–77
fscanf, 2–78
fseek, 2–79
fsetpos, 2–80
ftell, 2–81
fwrite, 2–82
getchar, 2–84
getenv, 2–85
getl, 2–86
gets, 2–87
getw, 2–88
gmtime, 2–89
isalnum, 2–90
isalpha, 2–91
iscntrl, 2–92
isdigit, 2–93
isgraph, 2–94
islower, 2–95
isprint, 2–96
ispunct, 2–97
isspace, 2–98
isupper, 2–99
isxdigit, 2–100
labs, 2–101
ldexp, 2–102
ldiv, 2–103
localeconv, 2–104
localtime, 2–105
log, 2–106
log10, 2–108
log2, 2–107
longjmp, 2–109
malloc, 2–110
mblen, 2–111
mbstowc, 2–113
mbstowcs, 2–112
memccpy, 2–114
memchr, 2–115
memcmp, 2–116
memcpy, 2–117
memmove, 2–118
memset, 2–119
mktime, 2–120
modf, 2–121
offsetof, 2–122
perror, 2–123
pow, 2–124
printf, 2–125
putc, 2–133
putchar, 2–134
putl, 2–135
puts, 2–136
putw, 2–137
qsort, 2–138
raise, 2–139
rand, 2–140
rcopy, 2–141
realloc, 2–142
remove, 2–143
rename, 2–144
rewind, 2–145
roupper, 2–195
scanf, 2–146
setbuf, 2–154
setjmp, 2–155
setlocale, 2–156
setvbuf, 2–157
signal, 2–158
sinh, 2–160
sprintf, 2–161
sqrt, 2–162
srand, 2–163
sscanf, 2–164
strcat, 2–165
strchr, 2–166
strcmp, 2–167
strcoll, 2–168

strcpy, 2–169
strcspn, 2–170
strerror, 2–171
strftime, 2–172
strlen, 2–174
strncat, 2–175
strncmp, 2–176
strncpy, 2–177
strpbrk, 2–178
strrchr, 2–179
strspn, 2–180
strstr, 2–181
strtod, 2–182
strtok, 2–183
strtol, 2–184
strtoul, 2–185
strxfrm, 2–186
swab, 2–187
system, 2–188
tan, 2–189
tanh, 2–190
time, 2–191
tmpfile, 2–192
tmpnam, 2–193
tolower, 2–194
ungetc, 2–196
va_arg, 2–197
va_end, 2–198
va_start, 2–199
vfprintf, 2–200
vprintf, 2–201
vsprintf, 2–202
wcstombs, 2–203
wctomb, 2–204

runtimelibrary routine, sin, 2–159

S

scanf, 2–146–2–153
SEGBASE, 4–27
SEGSIZE, 4–27
setbuf, 2–154
setjmp, 2–155
setlocale, 2–156
setvbuf, 2–157
sign extend instructions (EXT), 4–83
sin, 2–159
sinh, 2–160
sprintf, 2–161
sqrt, 2–162
srand, 2–163
sscanf, 2–164
statements
  assembler control, 3–5
  assembler directive, 3–4–3–5, 4–1
  data allocation, 3–4–3–5
  FOR, 7–8–7–18
  IF, 7–6–7–18
  instruction, 3–4–3–5
  REPEAT, 7–9–7–18
  source, 4–1, 4–3
  WHILE, 7–10–7–18
storage allocation, 2–11–2–32
strcat, 2–165
strchr, 2–166
strcmp, 2–167
strcoll, 2–168

strcpy, 2–169
strcspn, 2–170
strerror, 2–171
strftime, 2–172–2–173
strlen, 2–174
strncat, 2–175
strncmp, 2–176
strncpy, 2–177
strpbrk, 2–178
strrchr, 2–179
strspn, 2–180
strstr, 2–181
strtod, 2–182
strtok, 2–183
strtol, 2–184
strtoul, 2–185
strxfrm, 2–186
swab, 2–187
symbols, NARG, 6–6
system, 2–188
  initialization, 2–3–2–6

T

tan, 2–189, 2–203, 2–204
tanh, 2–190
test a logical address (PTEST), 4–106, 4–107, 4–108
time, 2–191
tmpfile, 2–192
tmpname, 2–193
tolower, 2–194
toupper, 2–195
trap on condition code (TRAP), 4–100

U

udata, 2–5
ungetc, 2–196
updating library, 2–16

V

va_arg, 2–197
va_end, 2–198
va_start, 2–199
validate a pointer (PVALID), 4–137
vfprintf, 2–200
vprintf, 2–201
vsprintf, 2–202